TESTING AND EXPOSING WEAK GRAPHICS PROCESSING UNIT MEMORY MODELS

by

Tyler Rey Sorensen

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2014

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Tyler Rey Sorensen**

has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan	, Chair	**5-30-2014** Date Approved
Zvonimir Rakamaric	, Member	**5-30-2014** Date Approved
Mary Hall	, Member	**5-30-2014** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Graphics Processing Units (GPUs) are highly parallel shared memory microprocessors, and as such, they are prone to the same concurrency considerations as their traditional multicore CPU counterparts. In this thesis, we consider shared memory consistency, i.e. what values can be read when issued concurrently with writes on current GPU hardware. While memory consistency has been relatively well studied for CPUs, GPUs present substantially different concurrency systems with an explicit thread and memory hierarchy. Because documentation on GPU memory models is limited, it remains unclear what behaviors are allowed by current GPU implementations.

To this end, this work focuses on testing shared memory consistency behavior on NVIDIA GPUs. We present a format for describing GPU memory consistency tests (dubbed GPU litmus tests) which includes the placement of testing threads into the GPU thread hierarchy (e.g. cooperative thread arrays, warps) and memory locations into GPU memory regions (e.g. shared, global). We then present a framework for running GPU litmus tests under system stress designed to trigger weak memory model behaviors, that is, executions that do not correspond to an interleaving of the instructions of the concurrent program. We discuss GPU specific incantations (i.e. heuristics) which we found to be crucial for observing weak memory model executions; these include bank conflicts and custom GPU memory stressing functions.

We then report the results of running GPU litmus tests in this framework and show that we observe a controversial relaxed coherence behavior on older NVIDIA chips. We present several examples of published GPU applications which may exhibit unintended behavior due to the lack of fence synchronization; one such example is a spin-lock published in the popular *CUDA by Example* book. We then test several families of tests and compare our results to a proposed operational GPU memory model and show that the model is unsound (i.e. disallows behaviors that we observe on hardware). Our techniques are implemented in a modified version of a memory model testing tool named litmus.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This work would not have been possible without the following people, to whom I extend my sincerest of gratitude.

First and foremost, thanks to my professor, mentor, and role model Professor Ganesh Gopalakrishnan for giving me the amazing opportunity to get involved in research. The experiences I've had over the last few years working with Professor Gopalakrishnan have given me a love for learning I never thought I could have. His tireless devotion to his students will not be forgotten. Thanks to my committee members, Professor Zvonimir Rakamaric and Professor Mary Hall. The mentoring, support, and opportunities they both have provided me have been essential in shaping my current interests and future goals.

Thanks to Dr. Jade Alglave at University College of London for supervising much of this work and her detailed feedback during the writing process. In addition to be being a knowledgeable and motivating mentor, she facilitated collaborations which gave this work breadth and momentum. Thanks to my UK GPU memory model collaborators, namely Daniel Poetzle (University of Oxford), Dr. Alastair Donaldson, Dr. John Wickerson (Imperial College London), Mark Batty (University of Cambridge), and Dr. Luc Maranget (Inria) for their insights and discussions that contributed to this work.

Thanks to Vinod Grover at Nvidia; his feedback and encouragement from an industry perspective helped steer us in new and interesting directions. Thanks to Professor Suresh Venkatasubramanian, Professor Stephen Siegel, and Professor Matt Might for their encouragement and invaluable contribution to my education over the last few years.

Thanks to my fellow aspiring researchers: Kathryn Rodgers, Mohammed Al-Mahfoudh, Bruce Bolick, and Leif Andersen for sharing my struggles and all their help, both academically and emotionally. Thanks to all the Gauss Group members for providing me with a stimulating environment and for forcing me to work harder than I ever have in my life trying to reach the precedence they have set. Thanks to my parents and other family members whose unwavering support and patience throughout my life has led me to where I am today. Lastly, and not exclusive from the aforementioned, thanks to my friends for their generous support throughout my education and consistent reminders that life is meant to be enjoyed.

I am grateful for the funding for this work which was provided by the following NSF awards: CCF 1346756, ACI-1148127, CCF-1241849, CCF 1255776, and CCF 7298529.

CHAPTER 1

INTRODUCTION

Much of the implementation work for this project was conducted during a three-month visit to University College London under the supervision of Dr. Jade Alglave. During that time, we met several other researchers interested in GPU memory models and began collaborating on a thorough study on the subject. This work presents one aspect of the larger study, namely running GPU litmus tests on hardware. However, this work was conducted in close collaboration with the larger project and draws heavy inspiration from discussions and work with the larger group, namely: Daniel Poetzl (University of Oxford), Dr. Alastair Donaldson, Dr. John Wickerson (Imperial College London), and Mark Batty (University of Cambridge).

A Graphics Processing Unit (GPU) is an accelerated co-processor (a processor used to supplement the primary processor often for domain-specific tasks) designed with many cores and high data bandwidth [1, pp. 3-5]. These devices were originally developed for graphics acceleration, particularly in 3D games; however, the high arithmetic throughput and energy efficiency of these microprocessors had potential to be used in other applications. In late 2006, NVIDIA released the first GPU that supported the CUDA framework [2, p. 6]. CUDA allowed programmers to develop general purpose code to execute on a GPU.

Since then, the use of GPUs has grown in many aspects of modern computing. For example, these devices have now been used in a wide range of applications, including medical imaging [3], radiation modeling [4], and molecular simulations [5]. Current research is developing innovative new GPU algorithms for efficiently solving fundamental problems in computer science, e.g. Merrill *et al.* [6] recently published an optimized graph traversal algorithm specifically to run on GPUs.

The most recent results (November 2013) of the TOP500 project, which ranks and documents the current most powerful 500 computers[1] in terms of performance, states that

[1] see `http://www.top500.org`

a total of 53 the computers on the list are using accelerators or co-processor technology, including the top two. A similar list known as the Green500[2] ranks super computers in terms of energy efficiency; GPU accelerated systems dominate this list and occupy all top ten spots.

Statistics from a popular online GPU research hub (`www.hgpu.org`) show how GPUs research has increased over the years. For example, less than 600 papers were published in 2009 describing applications developed for GPUs; in 2010 this rose to 1000 papers and years 2011 through 2013 each saw over 1200 papers. GPUs are also becoming common in the mobile market; popular tablets and smart phones, such as the iPad Air [7] and Samsung Galaxy S [8] series, now contain GPU accelerators.

GPUs are concurrent shared memory devices and as such, they share many of the concurrency considerations as their traditional multicore CPU counterparts including notorious concurrency bugs. One example of a concurrency bug is a *data race* in which shared memory is accessed concurrently without sufficient synchronization; data races cause undefined behavior in many instances (e.g. C++11 [9]). Another example of a concurrency bug is a *deadlock*, in which two processes are waiting on each other, causing the system to hang indefinitely. Concurrency bugs can be difficult to detect and reproduce due to the nondeterministic execution of threads. That is, a bug may appear in one run and not in another even with the exact same input data [10]. In some cases, concurrency bugs have gone completely undetected until deployment and have caused substantial damage. Notable examples include:

- The Therac-25 radiation machine, in which a data race caused at least six patients to be given massive overdoses of radiation [11].
- The Northeastern blackout of 2003, which left an estimated ten million people powerless for up to two days, was primarily due to a race condition in the alarm system [12].
- The 1997 Mars Pathfinder, in which a deadlock caused a total system reset during the first few days of its landing on Mars. Luckily the spacecraft was able to be patched from earth once the problem was debugged [13].

A related source of nondeterminism which can cause subtle and unintended (i.e. buggy) behaviors in concurrent programs is the *shared memory consistency model*, which is what values can be read from shared memory when issued concurrently with other reads and

[2] see `http://www.green500.org`

writes [14, p. 1]. A developer may expect every concurrent execution to be equivalent to a sequential interleaving of the instructions, a property known as *sequential consistency* [15]. This however, is not always the case as many modern architectures (e.g. x86, PowerPC, and ARM [16]) weaken sequential consistency for substantial performance and efficiency gains [17]. These architectures are said to have *weak memory models* and the underlying architecture is allowed to execute certain memory instructions out of the order in which they are given in the syntax of the program. We refer to executions that do not correspond to an interleaving of the instructions as *weak* or *relaxed* behaviors. To enable developers to enforce orderings not provided by the architecture, special instructions known as *memory fences* can be used to guarantee certain orderings and properties. If a programmer is to avoid costly and elusive concurrency bugs, he or she must understand the architecture's shared memory consistency model and the guarantees (or lack thereof) provided.

Shared memory consistency models for traditional CPUs have been relatively well studied over the years [14, 16, 18] and continue to be a rich area of research. However, GPUs have a hierarchical concurrency model that is substantially different from that of a traditional CPU. GPU developers have explicit access to the location of threads in the GPU thread hierarchy and can design programs using this information; threads that share finer grained levels of the hierarchy enjoy accelerated interactions and additional functionality. For example, one level of the hierarchy is called a *CTA* (Cooperative Thread Array). A GPU program often has many CTAs, and threads residing in the same CTA have access to a fast region of memory called *shared memory*[3]. Threads in different CTAs cannot access the same shared memory region and must use the slower *global memory* region to communicate data. Additionally, there are built-in synchronization barrier primitives and a memory fence that only apply to threads residing in the same CTA [19, p. 95]. These features are a noticeable departure from traditional CPU models where generally only one memory space is considered and memory fences apply to all threads.

Unfortunately, GPU vendor documentation on shared memory consistency remains limited and incomplete. The CUDA 6 manual provides only 3 pages of documentation on the subject, which largely covers memory model basics and shows one complicated example [19, pp. 92-95]. While NVIDIA does not release machine code documentation or tools, they provide a low-level intermediate language called PTX (Parallel Thread eXecution). The PTX 4.0 ISA gives only one page of shared memory consistency documentation with

[3]We use the term *shared memory* in this document to refer to the specialized GPU memory region as opposed to any region of memory that is accessible to multiple threads

no examples [20, p. 169]. Both CUDA and PTX documentation are written in prose and lack the mathematical rigor required to reason about complicated interactions. It remains unclear to us what behavior GPU developers can safely rely on when using current NVIDIA hardware.

1.1 Thesis Statement and Contributions

Due to the lack of a rigorous specification for the weak memory behaviors allowed by GPUs, it remains unclear what memory relaxations current GPUs allow. This issue can be systematically approached by developing formally-based testing methods that explore the behaviors observable on GPUs. These testing methods are able to experimentally investigate corner cases left underspecified by the documentation as well as rigorously test classic memory consistency properties (e.g. coherence); additionally this approach promotes the development of abstract formal models of the architecture, thus helping designers and developers agree on what user programs may rely upon. Without this understanding between designers and developers, GPU applications may be prone to elusive bugs due to weak memory orderings. While these testing approaches have been employed successfully for CPU architectures, GPUs contain an explicit hierarchical concurrency model with subtle scoped properties unseen on CPU architectures; additionally, the throughput oriented hardware of GPUs require innovative new testing heuristics in order to effectively reveal weak behaviors.

1.1.1 Thesis Statement

Systematic memory model explorations are greatly aided by developing formally-based testing methods that reveal experimentally the extent to which the memory orderings are relaxed. In addition to helping corroborate with intentionally designed relaxations, these approaches also help expose unintended weak behaviors (bugs), and also help set allowed weakenings for the architectural family.

1.1.2 Contributions

To better understand and test GPU memory models, this work presents a GPU hardware memory model testing framework which runs simple concurrent tests (known as *litmus tests*) thousands of times under complex system stress designed to trigger weak memory model behavior. The results are recorded and checked for weak memory model behaviors and how often they occurred. We present a format for describing GPU litmus tests which account for the explicit placement of threads into the GPU thread hierarchy and memory locations

into GPU memory regions. The framework reads a GPU litmus test and creates executable CUDA or OpenCL code with inline PTX which will run the test and display the results.

We develop GPU-specific heuristics without which we are unable to observe many weak model behaviors. These heuristics include purposely placing poor memory access patterns (known as *bank conflicts*) on certain memory accesses in the tests and randomly placing the testing threads throughout the GPU. For example, if the GPU litmus test specifies two testing threads are in different CTAs, the framework will then randomly assign a distinct CTA ID to the testing thread for each run of the test. Our testing framework also uses the nontesting threads on the GPU to create memory stress by constantly reading and writing to nontesting memory locations. These heuristics have a substantial impact on if, and how many times, weak behaviors are observed.

We then report the results of running GPU litmus tests in this framework. We observe a controversial and unexpected relaxed coherence behavior, in which a read instruction is allowed to observe stale data w.r.t. an earlier read from the same address. We observe this behavior on older NVIDIA chips, but not the newest architecture (named Maxwell). We present several examples of published GPU applications which may exhibit unintended behavior due to the lack of fence synchronization. These examples include a spin-lock published in the popular *CUDA by Example* book and a dynamic GPU load balancing scheme published as a chapter in *GPU Computing GEMs - Jade Edition*. We test many classical CPU litmus tests under different GPU configurations and show that GPUs implement weak memory models with subtle scoped properties unseen in CPU models. Finally, we compare our testing results to a proposed operational GPU memory model and show that it is unsound, i.e. disallows behaviors that we observe on hardware.

Our techniques are implemented in a modified version of the litmus tool of the DIY memory model testing tool suite (see `http://diy.inria.fr/`).

1.2 Prior Work

The work presented in this thesis draws heavy inspiration from the original litmus tool [21] of the DIY memory model testing tool suite[4] which runs litmus tests on several different CPU architectures, including x86, PowerPC, and ARM. It takes a litmus test written in pseudo assembly code as input and creates executable C code which will execute and record the outcomes of the input litmus test. The litmus tool uses heuristics to make weak behaviors show up more frequently which include affinity assignments and

[4]see `http://diy.inria.fr/`

custom synchronization barriers. The work presented in this thesis modifies the litmus tool to take GPU litmus tests as input and creates executable CUDA or OpenCL code with GPU-specific heuristics. TSOTool [22] is another memory model testing tool which exclusively targets architectures which implement the Total Store Order (TSO) memory model. The ARCHTEST tool [23] is an earlier memory model testing tool which only tests for certain behaviors and cannot easily run new tests as the tests are hard coded in the tool.

Using litmus tests are an intuitive way to understand memory consistency models and are used in official industry documentation [24]. Litmus tests have been studied formally and have been shown to describe important properties of memory systems such as model equivalence [25]. Alglave *et al.* have developed a method for generating litmus tests [26] based on cycles and present large families of litmus tests in [16]. This thesis expands the traditional CPU litmus test with additional GPU unique specifications (described in Section 3.1).

1.2.1 GPU Memory Models

The past two years have seen a noticeable push in both academia and industry to understand and document GPU memory models. We consider this work part of that effort and hope to see same level of rigorous testing and modeling applied to GPU memory models as CPU memory models have enjoyed (for example, in [16, 18, 14]).

We present here the history as we know it of GPU memory models in prior literature and how this work on testing contributes to them:

- In June 2010, Feng and Xiao revisited their GPU device-wide synchronization method [27] to repair it with fences [28]. They report on the high overhead cost of GPU fences, which in some cases removes the performance gain of their original barrier. They appear skeptical that GPUs exhibit weak memory behaviors, illustrated by the following quote [28, p. 1]:

  > In practice, it is infinitesimally unlikely that this will ever happen given the amount of time that is spent spinning at the barrier, e.g., none of our thousands of experimental runs ever resulted in an incorrect answer. Furthermore, no existing literature has been able to show how to trigger this type of error.

  We consider our work to be a response to that quote in that we show heuristics which trigger weak memory effects on GPUs (see Chapter 3).

- In June 2013, Hower *et al.* proposed a SC (Sequential Consistency) for race-free memory model for GPUs [29]. This model uses acquire/release [14, pp. 68–69] syn-

chronization; however, to allow efficient use of the explicit GPU thread hierarchy, the acquire and release atomic operations may be annotated with a scope (i.e. level) in the GPU hierarchy which restricts the ordering constraints to that scope. Using these atomics and program order, they construct a happens-before relation which they use to define a particular type of data race they dub a *heterogeneous data race*. They state that hardware satisfying this memory model must give sequentially consistent behavior for programs free of heterogeneous data races. While this model is intuitive, it is unclear if or how this is to be implemented on current hardware.

- Also in June 2013, work by Hechtman and Sorin [30] showed that in a particular model of GPU and for programs run on GPUs, weak memory consistency has a negligible impact on performance and efficiency. Because of this, the authors suggest that sequential consistency is an attractive choice for GPUs. In our work, we show that regardless of the benefits (or lack thereof) of weak memory consistency on GPUs, current GPUs do in fact implement weak memory models.

- Continuing in June 2013, Sorensen *et al.* [31] proposed an operational weak GPU memory model based on the limited available documentation and communication with industry representatives. This model was implemented in a model checker and gave semantics to simple scoped GPU fences over shared and global memory regions. More complex interactions were left unspecified. In our work (Section 5.4), we compare the behaviors allowed under this model against behaviors observed on hardware and show that this model is unsound (i.e. the model disallows behaviors that we observe on hardware).

- In January 2014, Hower *et al.* [32] continued their work and present two SC for data race free GPU memory models using scoped acquire/release semantics again. The first model, dubbed HRF-direct, is suited for traditional GPU programs and current language standards model. The second model, dubbed HRF-indirect, is forward-looking to irregular GPU programs and new standards. Much like their previous work in [30], this work describes intuitive models, but it still remains unclear if or how this relates to memory models on current GPUs.

At this point, we have only discussed NVIDIA specific industry documentation. However, non-NVIDIA proprietary GPU languages and frameworks have begun to explore GPU memory models. The new OpenCL 2.0 [33] GPU programming language specification released in November of 2013 has adopted a memory model similar to C++11 [9]. However,

to enable developers to take advantage of the explicit GPU thread hierarchy, the OpenCL 2.0 specification has introduced new memory scope annotations to atomic operations which restricts ordering constraints to certain levels in the GPU thread hierarchy. Similarly, the HSA low-level intermediate language [34] provides scoped acquire/release memory operations and fences similar to the previously mentioned work by Hower *et al.* [32]. Our work empirically investigates the current GPU hardware memory models, which must be well understood if these new specifications are to be efficiently implemented.

1.3 Roadmap

Chapter 2 presents the required background for the proper understanding of the rest of this document. This includes a primer on GPU architectures and programming models including the relevant low-level PTX instructions. Furthermore, we discuss some prerequisites on shared memory consistency and litmus tests. We conclude this chapter by formally discussing our notation for GPU litmus tests.

In Chapter 3, we discuss our testing framework, starting with the format of a GPU `.litmus` test for the PTX architecture. We then discuss *critical incantations*, without which we are unable to observe any weak memory model behaviors. We continue to present additional heuristics and report on their effectiveness.

Chapter 4 presents several notable results that we have gathered from running tests with the framework. We show a controversial relaxed coherence behavior observable on older NVIDIA GPUs, but not on the most recent architecture. We discuss interesting behaviors with PTX memory instruction annotations, and show examples where we observed behaviors that we did not expect from reading the documentation. We then shift our focus to CUDA applications (two of them published in CUDA books) which contain interesting concurrent idioms, namely two mutex implementations and a concurrent data structure. We show that these implementations may allow unintended (i.e. buggy) executions but may be experimentally repaired with memory fences.

In Chapter 5, we present the results of running families of different tests under several GPU configurations. We show that GPUs implement weak memory models with subtle scoped properties unseen in CPU models. These families of tests provide intuition about what types of re-orderings are allowed on GPUs and what memory fences will experimentally restore orderings. We compare our observations to an operational GPU model presented in [31] and show that the model is unsound (i.e. disallows behaviors that we observe on hardware).

We end with a conclusion in Chapter 6 which discusses ongoing work and future work. Specifically, we discuss different GPU configurations that we were unable to test in this document and interesting results they could yield. Additionally, we show new features being added to the Herd [16] axiomatic memory model framework to reason about GPU memory models. We finish with a summary of the document.

CHAPTER 2

BACKGROUND

In this chapter, we discuss the necessary background required for this work, including an overview of GPU programming and hardware models (in Section 2.1 and 2.2, respectively). Section 2.3 discusses the NVIDIA low-level intermediate PTX language and the instructions we consider in this document. We provide a table of CUDA to PTX compilation mappings in Section 2.3.1 which enables us to reason about CUDA code using PTX test cases. Section 2.4 then contains a primer on memory consistency models and litmus tests. We more formally define the litmus test format, naming conventions and GPU configurations we consider in Section 2.5.

Different GPU frameworks and vendors use different terminology and often overload terms that have established meanings in traditional concurrent programming (e.g. *shared memory*). Because this work focuses largely with NVIDIA GPU hardware, we use similar terminology to that in the PTX ISA [20]. Table 2.1 shows a mapping from other GPU terminologies to the ones we use; recall HSA is a new standard for heterogeneous computing, including GPUs [34].

2.1 GPU Programming Model

Programs that execute on GPUs are called GPU *kernels* and consist of many threads which are partitioned in the GPU thread hierarchy. Threads that share finer grained levels of the hierarchy have additional functionality which developers can design their GPU kernels

Table 2.1. GPU terminology mappings between different vendors and frameworks

PTX	**CUDA**	**OpenCL**	**HSA**
thread	thread	work-item	work-item
warp	warp	subgroup	wavefront
CTA	thread-block	work-group	work-group
shared memory	shared memory	local memory	group memory
global memory	global memory	global memory	global memory

to exploit. There are four levels of the GPU thread hierarchy that are considered in this work:

- **Thread**: Much like a CPU thread, a GPU thread executes a sequence of instructions specified in the GPU kernel.
- **Warp**: For all available NVIDIA architectures, a warp consists of 32 threads. Threads in the same warp are able to quickly perform reductions and share variables via the warp vote and warp shuffle CUDA function [19, pp. 114–118].
- **CTA**: A *Cooperative Thread Array* or CTA consists of a variable number of warps which can be programmed at run-time. Depending on the GPU generation, a CTA can contain up to 16 or 32 warps (512 or 1024 threads). Threads in the same CTA are able to efficiently synchronize via a built-in synchronization barrier called with the `__syncthreads` command in CUDA [19, pp. 95–96].
- **Kernel**: A kernel (or GPU program) consists of a variable number of CTAs, which may be in the millions. Distinct CTAs share the slowest memory region (global memory) and have very limited support for interacting. There is no synchronization barrier for all CTAs; however, there is a memory fence [19, p. 93] and read-modify-write atomics [19, p. 111] which are supported to work across distinct CTAs. It should be noted that CTAs are not guaranteed to be scheduled concurrently and deadlocks may occur if a CTA is waiting for another CTA that is not scheduled [19, p. 12].

In addition to the functionality available at different levels of the GPU hierarchies, GPUs also provide different memory regions that are only shared between threads in common hierarchy levels. These memory regions are:

- **Global Memory**: This region of memory is shared between all threads in the GPU kernel.
- **Shared Memory**: This region of memory is shared only between threads in the same CTA; it is considerably faster and smaller than the global memory region.

Many GPUs additionally provide read-only memory regions (e.g. known as *constant* and *texture* memory in CUDA). These memory regions are not considered in this work because they are uninteresting with respect to shared memory consistency, i.e. the set of values a read can return from read-only memory region is simply the memory value with which it was initialized. The GPU thread and memory hierarchy are shown in Figure 2.1.

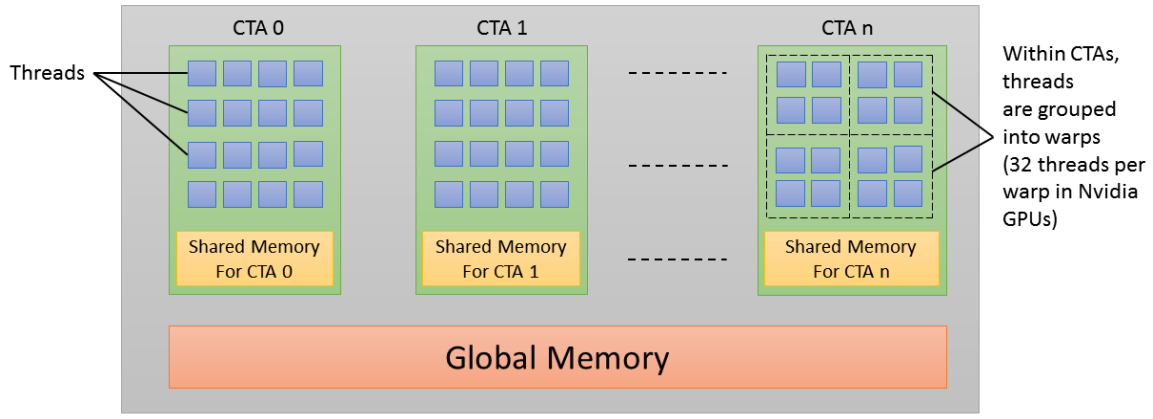

Figure 2.1. GPU thread and memory hierarchy of the GPU programming model

GPU kernels are written as a single function which all threads in the kernel execute. Threads are able to query special variables (or registers in PTX) to determine the ID of the CTA to which they belong, the size of their CTA, and their thread ID within the CTA. Using this information, threads are able to compute a unique global ID and can then access unique data to operate on. For example, a GPU kernel to add two vectors `x` and `y` and store the result in vector `z` written in CUDA is shown in Figure 2.2. This program assumes that the kernel has exactly as many threads as elements in the vector.

A GPU kernel is called from a CPU function using triple chevron style syntax, where the two arguments inside the chevrons are the number of CTAs and threads per CTA. For example, to launch the GPU kernel shown in Figure 2.2 with `c` CTAs and `t` threads per

```
1  //__global__  specifies that this function starts a GPU kernel
2  __global__ void add_vectors(int *x, int *y, int *z) {
3
4     int cta_id = blockIdx.x;     //special variable for cta ID
5     int cta_size = blockDimx.x;  //special variable for cta size
6     int thread_id = threadIdx.x; //special variable for thread ID
7
8     //A unique global ID can be computed from the above values as:
9     int global_id = (cta_id * cta_size) + thread_id;
10
11    //Now each thread adds its own array index
12    z[global_id] = x[global_id] + y[global_id];
13 }
```

Figure 2.2. Vector addition GPU kernel written in CUDA

CTA would be written as: `add_vectors<<<c,t>>>(x,y,z);`. Finally, the CPU may not directly access GPU memory; it must be explicitly copied to and from the GPU through a built-in CUDA function named `cudaMemCpy`.

2.2 GPU Architecture

The GPU hardware architecture consists of physical processing units and a cache hierarchy onto which the programming model maps. The architecture white papers published by NVIDIA provide detailed information about the different features of the hardware. In this document, we focus on the Fermi, Kepler, and Maxwell (GTX 750 Ti) architectures, whose white papers are [35], [36], and [37], respectively.

A GPU consists of several *streaming multiprocessors* (or SMs). Larger GPUs designed for HPC and heavy gaming may contain up to 15 SMs (e.g. GTX Titan) while smaller GPUs may have much fewer; for example, the GTX 540m GPU has only 3 SMs. Each SM contains a number of CUDA cores with pipelined arithmetic and logic units. The Fermi architecture contains 32 CUDA cores per SM while the Kepler architecture features 192 CUDA cores per SM. All threads in the same CTA are mapped to CUDA cores in the same SM and are executed in groups of 32 (i.e. a warp) in a model known as single instruction, multiple threads (or *SIMT*) [19, pp. 66–67]. In this model, all threads in the warp are given the same instruction to execute similar to the SIMD model in Flynn's taxonomy [38]. However, the SIMT model differs from the SIMD model in that all threads have unique registers and not all threads must execute the instruction (e.g. if a conditional only allows some threads of a warp into a program region, then the other threads in the warp simply do not execute until the conditional block of code ends). The Fermi architecture has a dual warp scheduler that may issue instructions from two independent warps concurrently while the Kepler architecture features a quad warp scheduler. The maximum number of threads that can be assigned to an SM at any given time is 1536 and 2048 for Fermi and Kepler, respectively. GPUs are attached to the main CPU through the PCI bus.

GPUs contain a physical cache hierarchy for the memory regions of the programming model to map onto. A GPU contains a large region of DRAM to which all SMs have access; it houses global and constant memory. This memory is usually 1 to 6 GBs in size. The entire GPU then shares an L2 cache which is typically 1 to 2 MB in size and accelerates global and constant memory accesses. Each SM contains a region for shared memory and also a L1 cache for global and constant memory. In the Fermi and Kepler architectures, this region of memory is the same and developers are free to configure this region to have

more shared memory or more L1 cache. In the Maxwell architecture, the shared memory region and L1 cache are separate. This region of memory is typically 64 KB in size. It is documented that the L2 cache is coherent (see Section 4.2 for a discussion of coherence), but multiple interacting L1 caches are *not* coherent, e.g. two SMs accessing global memory via their respective L1 caches are not guaranteed to have coherent interactions. GPU memory instructions can be annotated to enforce which cache is targeted; these annotations are documented in Section 2.3.

A figure of the GPU hardware model is shown in Figure 2.3. Notice the similarities to the programming model shown in Figure 2.1, i.e. threads map to CUDA cores, CTAs map to SMs, shared memory maps to the L1/shared memory cache, and global memory maps to the L2/DRAM memory.

2.2.1 Hardware Memory Banks

One aspect of the GPU architecture that is used in this work is the different ways that the hardware handles concurrent memory accesses. The shared memory region on a GPU is organized in 32 4-byte banks on each SM [39, p. 118]. When threads in a warp issue a

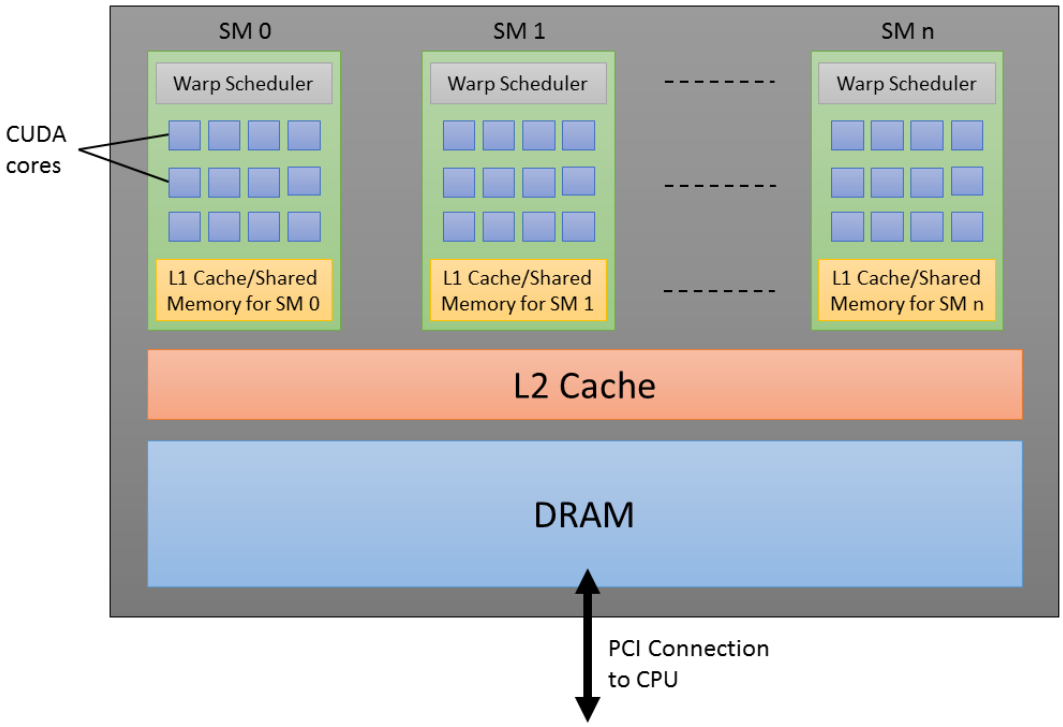

Figure 2.3. GPU hardware showing CUDA cores, SMs, and the memory hierarchy

memory access from shared memory, three things may happen which are shown in Figure 2.4 and described below:

- **Parallel Access**: In a parallel access, each thread in the warp accesses a unique hardware bank and memory requests are able to be serviced in parallel.
- **Broadcast**: In a broadcast access, only one memory load is issued and the result is broadcast to all threads. This access only applies to load operations and happens when threads load from the same address.
- **Bank Conflict**: In a bank conflict access, the hardware serializes the accesses which causes a performance slowdown. This access is similar to a broadcast access except that threads access different addresses from the same bank.

Additionally, GPUs are sensitive to the alignment of global memory accesses. Cache lines are 128 bytes, and warps that access across multiple cache lines result in unnecessary data movement (i.e. entire cache lines) which causes a loss of performance. Avoiding these types of poorly aligned accesses is known as *memory coalescing* [39, pp. 125–127].

2.3 PTX

We have previously mentioned that GPUs may be programmed using CUDA language; however, the goal of this work is to test GPU hardware, and as such it is convenient to program as close to the hardware as possible. The CUDA compilation process takes a file

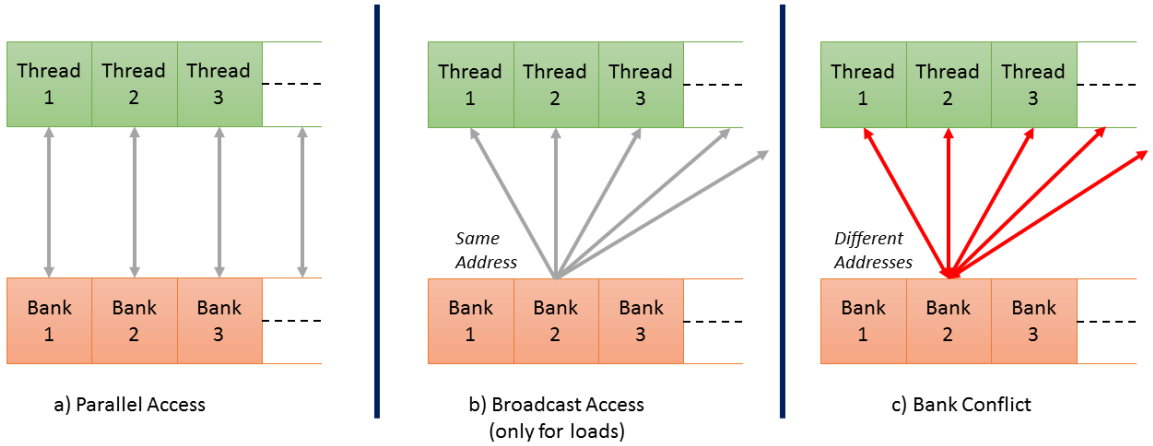

Figure 2.4. Different types of concurrent memory accesses within a warp: a) parallel access where threads reads different banks, b) broadcast access where threads read from the same bank and same address, and c) bank conflict access where threads access the same bank but different addresses

with a program written in the CUDA language as input and compiles it into a GPU binary file known as a *cubin* which contains GPU machine code. As part of this process, a low-level intermediate representation known as Parallel Thread eXecution (or PTX) is generated.

NVIDIA provides very limited access to the machine code, which is very sparsely documented [40]. Additionally, there is no available method to write inline GPU machine code or even assemble machine code programs. The sole access to GPU machine code is through the application `cuobjdump` which provides the assembly code of a `cubin` file. To this end, our framework tests the hardware by using inline PTX in CUDA or OpenCL code which is supported [41].

PTX syntax requires each instruction to contain a type annotation specifying the data type the instruction is targeting. For example, an unsigned 32 bit type carries an annotation of `.u32`. Additionally, memory instructions may be annotated to specify different caching behaviors. For example, a load instruction (`ld`) may be annotated to read from the L2 cache with annotation `.cg`. As a complete example, to load an unsigned 32 bit value from the L2 cache, the following instruction would be used: `ld.cg.u32`. Table 2.2 shows the types, annotations, and instructions that this work targets with a brief description interpreted from the PTX ISA [20] to the best of our understanding.

2.3.1 CUDA to PTX Mappings

In Chapter 4, we discuss several case studies where we evaluate published CUDA code in our testing framework. Because our framework evaluates PTX code, CUDA instructions must first be mapped to PTX instructions. Table 2.3 shows the relevant instruction mappings from CUDA to PTX for these case studies which we have discovered by examining CUDA code and generated PTX code[1]. We have taken precautions to ensure that loads and stores are compiled with the L2 memory annotation. This is done because Section 4.3 shows that is not possible to restore orderings to operations that target the L1 cache (the default for the CUDA compiler) on the Fermi architecture. We are interested in experimentally examining which fences are required to restore orderings to the examples, thus instructions to which we are unable to restore order are not interesting. The L2 annotation can be set to the default with the following compiler flags: `-Xptxas -dlcm=cg -Xptxas -dscm=cg`.

The focus of this document is to show the behaviour of these examples at the hardware level; as such, we ignore the effects of potential compiler optimizations. For the CUDA case studies we examine, we have verified by manually inspecting the PTX output that the CUDA

[1]We used `CUDA release 5.5 V5.5.0`

Table 2.2. Relevant PTX data types, memory annotations, and instructions

PTX Data Types	
`.u32`	unsigned 32 bit integer type
`.s32`	signed 32 bit integer type
`.b32`	generic 32 bit type
`.b64`	generic 64 bit type
`.pred`	predicate (contains either true or false)
PTX Memory Operation Annotation	
`.ca`	annotates load instructions, loads values from the L1 cache
`.wb`	annotates store instructions, stores values to L2 cache, but future architectures may use it to store to L1 cache
`.cg`	annotates both load and store instructions, accesses will target L2 cache
`.volatile`	annotates both load and store instructions, inhibits optimizations and may be used to enforce sequential consistency
PTX Instructions	
`ld`{*.ann*}{*.type*} `r1, [r2]`	loads value at address in register `r2` into register `r1` of data type *type* with annotation *ann*
`st`{*.ann*}{*.type*} `[r1], r2`	stores value in register `r2` of data type *type* to the address in register `r1` with annotation *ann*
`membar`{*.scope*}	memory fence for *scope* of `.cta` or `.gl` for inter-CTA and interdevice, respectively
`atom`{*.op*}{*.type*} `r1, [r2], r3`	atomically perform operator *op* with memory at address `r2` and value in register `r3` and stores the previous memory value in register `r1`. *op* may be `.add` to atomically add or `.exch` to exchange etc.
`setp`{*.comp*}{*.type*} `p1, r1, r2`	sets the value of the predicate register `p1` to the value of comparing registers `r1` and `r2` with *comp* where comp might be `.gt` (greater than), `.eq` (equal to) etc.
PTX Predicates	
`@p1` {*ins*}	execute instruction *ins* only if predicate register `p1` is true

Table 2.3. CUDA compilation mappings to PTX

CUDA Instruction	PTX Instruction
`atomicCAS`	`atom.cas.b32`
`atomicExch`	`atom.exch.b32`
`__threadfence`	`membar.gl`
`__threadfence_block`	`membar.cta`
store global int	`st.cg.u32`
load global int	`ld.cg.u32`
store global volatile int	`st.volatile.u32`
load global volatile int	`ld.volatile.u32`
control flow (e.g. `while`, `if`)	`setp` with predicate (e.g. `@r1`)

compiler does not reorder or otherwise optimize the memory accesses (e.g. hold memory accesses in registers). For PTX tests, we have again manually inspected the assembly code, using `cuobjdump`, to ensure that the PTX compiler does not reorder or otherwise optimize the memory accesses; future work will attempt to automate this validation. Because of this manual work, we can ignore compiler optimizations for the examples we present and be sure that we are indeed testing the hardware behavior.

2.4 Memory Models and Litmus Tests

For a given program and architecture, a memory model defines the set of values that the load instructions are allowed to return. That is, it specifies all possible behaviors of shared memory interactions. Memory models may be described in an operational style in which the system is described as an abstract machine. Given the current state of the system, the operational model will provide all possible transitions the system could take and how the system state is updated based on the transition; examples of operational models include [42, 18]. Memory models may alternatively be defined in an *axiomatic* style where constraints are described on sets and relations over memory actions; for examples of this type of model, see [43, 44, 16]. Our work does not propose any memory model; instead, we examine the observable effects of the memory model implemented on current GPUs. In Section 5.4, we compare our results to a proposed operational GPU memory model and show that the model is unsound (i.e. disallows behaviors that we observe on hardware). In Section 6.2, we briefly discuss future work to extend the herd axiomatic memory model tool [16] of the DIY tool suite for GPU memory models.

An intuitive way to understand memory models is through *litmus tests*, i.e. short concurrent programs with an assertion about the final states of registers and memory. Litmus

tests are evaluated under a memory model and can be allowed (the assertion sometimes passes) or disallowed (the assertion never passes). Figure 2.5 shows a litmus test known as *store buffering* (or abbreviated to SB) written in C-like syntax. In this test, `x` and `y` are memory locations initialized to 0. Thread 0 first stores the value `1` to location `x` then reads from location `y` and stores the result in local register `r1`. Thread 1 writes to location `y` and then reads from `x` and stores the result in local register `r2`. The assertion asks if `r1` and `r2` are allowed to both equal `0` after both threads finish executing.

Many programmers are taught to reason about concurrent programs under the *sequentially consistent* memory model (or simply SC), first defined by Lamport in 1979 [15]. That is, a concurrent execution must correspond to some interleaving of the instructions. Figure 2.6 shows how one would reason about the SB litmus test (shown in Figure 2.5) under SC; that is, the interleavings are enumerated and executed as a sequential program. There are six possible interleavings and the assertion ($r1 = 0 \wedge r2 = 0$) is not satisfied for any of them, thus the SB litmus test is disallowed under the SC memory model.

Modern multiprocessors (e.g. x86, ARM) implement *weak memory models*, where executions may not correspond to an interleaving. Using the original litmus tool [21] to run the store buffering litmus test on an Intel i7 processor one million times yields the histogram of results shown in Figure 2.7 (the output has been slightly modified from the actual litmus output to correspond to the register syntax used throughout in this section). This shows that empirically we can observe that the Intel i7 processor allows weak behaviors (executions that do not correspond to an interleaving of the instructions) in 119 out of a million iterations.

Weak architectures provide *fence* instructions to restore orderings. For example, considering the store buffering litmus test shown in Figure 2.5, if we place the x86 fence instruction `mfence` between instructions `a` and `b` and instructions `c` and `d` and execute the test again, we do not observe any weak behaviors and the litmus test becomes disallowed.

initial state: x = 0, y = 0	
Thread 0	Thread 1
`a: x ← 1;`	`c: y ← 1;`
`b: r1 ← y;`	`d: r2 ← x;`
assert: $r1 = 0 \wedge r2 = 0$	

Figure 2.5. Store buffering (SB) litmus test

Interleaving 1	**Interleaving 2**	**Interleaving 3**
a: x ← 1;	a: x ← 1;	a: x ← 1;
b: r1 ← y;	c: y ← 1;	c: y ← 1;
c: y ← 1;	b: r1 ← y;	d: r2 ← x;
d: r2 ← x;	d: r2 ← x;	b: r1 ← y;
Final: r1 = 0 ∧ r2 = 1	Final: r1 = 1 ∧ r2 = 1	Final: r1 = 1 ∧ r2 = 1
Interleaving 4	**Interleaving 5**	**Interleaving 6**
c: y ← 1;	c: y ← 1;	c: y ← 1;
a: x ← 1;	a: x ← 1;	d: r2 ← x;
b: r1 ← y;	d: r2 ← x;	a: x ← 1;
d: r2 ← x;	b: r1 ← y;	b: r1 ← y;
Final: r1 = 1 ∧ r2 = 1	Final: r1 = 1 ∧ r2 = 1	Final: r1 = 1 ∧ r2 = 0

Figure 2.6. All interleaving of the store buffering (SB) litmus test

2.5 GPU Litmus Tests

Here we formally define our notation for the presentation of GPU litmus tests and show a concrete example of a PTX litmus test. Additionally, we present the three different GPU configurations on which we focus throughout this document.

A *test specification*, such as the one shown in Figure 2.8, consists of several columns, each headed by a global thread ID. Each thread scheduled on the GPU has a unique global thread ID. In practice, a global thread ID can be computed using a combination of the built-in GPU values, i.e. thread ID, CTA ID, and CTA size. However, in our examples, we use symbolic global thread IDs, such as T0 and T1, for ease of presentation. A brief description of each major part of the test specification follows.

```
Test SB Allowed
Histogram (4 states)
119    *> r1=0; r2=0;
499580 :> r1=1; r2=0;
500248 :> r1=0; r2=1;
53     :> r1=1; r2=1;
Ok

Witnesses
Positive: 119, Negative: 999881
Condition exists (r1=0 /\ r2=0) is validated
```

Figure 2.7. Histogram of results from running the store buffering litmus test on an Intel i7 x86 processor.

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`ld.cg.s32 r1, [y] ;`
`st.cg.s32 [y], 1 ;`	`ld.cg.s32 r2, [x] ;`
assert: 1:r1=1 ∧ 1:r2=0	

Figure 2.8. Litmus test example written for GPUs in PTX syntax

In each test, the initialization of memory will be explicitly provided at the top of the test. In the example shown in Figure 2.8, the memory locations are initialized to 0.

Under each global thread ID is a *program*, i.e. the sequence of instructions executed by that thread. In GPU programming, every thread executes the same program; however, we can arrange for each thread to execute a different program by having threads branch to different parts of the program based on their global thread IDs (we discuss this more fully in Section 3.2). Consider the example in Figure 2.8, which implements a message passing idiom and is known as MP (this test is analyzed in Section 5.2.1; it is given here for explanatory reasons only). Here, the global thread IDs are T0 and T1. We assume that each kernel is launched with a sufficient number of CTAs and threads such that each program in the test will eventually be executed on the GPU. In the example, the two store instructions will be executed by thread T0, and the two load instructions will be executed by thread T1.

We deviate from concrete PTX syntax in that we allow direct stores of immediate values to memory (e. g. `st.cg.s32 [x],1` as seen in T0's program in Figure 2.8), instead of moving the value first to a register (via a `mov` instruction), and then storing the register contents to memory. Thread local registers are denoted by `r`n where n is a non-negative integer. Locations are given by single lower-case letters, e. g. `x,y,z`. In Figure 2.8, there are two memory locations `x` and `y`, and thread `T1` loads memory values to registers `r1` and `r2`.

Questions about the executions of a test are given as an assertion on the final values of registers or memory locations. In Figure 2.8, the constraint is given as:

Assert: 1:r1=1 ∧ 1:r2=0

to ask if it is possible to observe T1's private registers `r1` to be 1 and `r2` to be 0 in the final state of the GPU after having executed all testing threads. Here, registers in the final constraint are denoted n:`reg`, where `reg` is a register and n is the ID of the thread to which the register belongs.

2.5.1 GPU Configurations

Regarding Figure 2.8, the test may yield different behaviors depending on whether T0 and T1 are in the same CTA or in different CTAs. Similarly with memory regions, the behaviors allowed may be different depending on which GPU memory region `x` and `y` are located in (shared or global). We refer to the placement of testing threads into the thread hierarchy and memory locations into memory regions as a *GPU configuration*.

Although in Section 3.1 we show that our testing framework can execute most GPU configurations, in this document, we largely only consider three simple GPU configurations. We refer to them as D-warp:S-cta-Shared, D-warp:S-cta-Global, and D-cta:S-ker-Global; they are defined as follows:

- **D-warp:S-cta-Shared**: In this GPU configuration, all programs in the test are mapped to threads in different warps (D-warp stands for *different warps*), but in the same CTA (S-cta stands for *same CTA*). Additionally, all testing memory locations are located in the shared memory region.
- **D-warp:S-cta-Global**: Similar to D-warp:S-cta-Shared, in this GPU configuration, all programs in the test are mapped to threads in different warps (D-warp), but in the same CTA (S-cta). However, all testing memory locations are located in the global memory region.
- **D-cta:S-ker-Global**: In this GPU configuration, all programs in the test are mapped to different CTAs (D-cta) but the same kernel (S-ker). There is no shared memory region variant of this GPU configuration because threads in different CTAs have disjoint shared memory regions.

2.5.1.1 Limitations

We note that these are not a complete set of GPU configurations. For example, in 3+ threaded tests, some threads may be in the same CTA and others may be in different CTAs. Similarly the same test may contain both shared and global memory locations. However, because the three configurations we examine are explicitly discussed in the PTX ISA [20, p. 165], we believe these configurations serve as a good basis for our exploration. In ongoing work discussed in Section 6.1 we consider more complicated GPU configurations, which show interesting initial results.

In this document, we do not test intrawarp interactions. This is largely because of the essential role that warps have in our testing frameworks incantations, supporting intrawarp

testing becomes very difficult to develop and maintain. For example, in the synchronization incantation described in Section 3.4.2, we note that only one thread per-warp can execute the synchronization barrier; if multiple threads within a warp are executing tests with the synchronization incantation, then the high-level kernel design (shown in Section 3.2) will deadlock, and a much more complicated high-level design will be needed. We instead chose to spend our energy developing other features, such as read modify write atomics and conditionals, which produced interesting results as seen in Chapter 4.

Some applications contain multiple kernels which run on multiple GPUs concurrently. In this document, we do not consider multi-GPU interactions because different GPU chips may implement different memory models, e.g. in Chapter 5, it can be seen that different GPU chips experimentally allow different behaviors. One of the goals of this work is to provide empirical benchmarks to compare putative GPU memory models against. Composing different memory models deserves careful treatment, and given that we do not even have a memory model for single GPU interactions, we believe such a study is outside the scope of this document.

CHAPTER 3

GPU TESTING FRAMEWORK

In this chapter, we discuss in detail the GPU testing framework. The high-level flow of the framework is shown in Figure 3.1. First, the GPU litmus tool is given a GPU litmus test in the GPU `.litmus` format which we describe in Section 3.1. This test specification is used to create CUDA or OpenCL code which can be compiled and executed on a GPU. The program will create a histogram of the results of running the test many times and check if any of the outcomes satisfied the assertion given in the GPU `.litmus` file.

Most of this section is devoted to discussing the GPU program which is produced by the litmus tool and the heuristics (we dub incantations) that we use to expose weak behaviors. Section 3.3 presents *critical incantations*, which are the incantations without which we are unable to observe any weak behaviors. Section 3.4 presents additional heuristics which greatly increases the number of weak behaviors we are able to observe. We end this chapter by showing the effectiveness of these heuristics in Section 3.5.

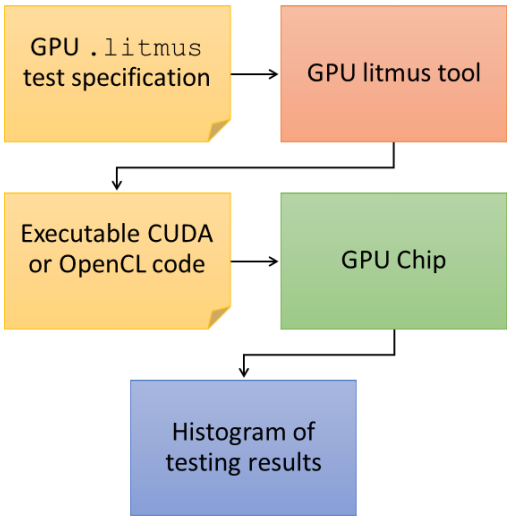

Figure 3.1. High-level flow of the GPU litmus tool

3.1 PTX GPU `.litmus` Format

Figure 3.2 shows a complete example of PTX GPU `.litmus` test which is able to be parsed by the litmus tool to produce CUDA or OpenCL code. This test encodes the store buffering (SB) litmus test first discussed in Section 2.4. We proceed to discuss each section of the GPU `.litmus` test in detail. We note that the style and syntax of the GPU `.litmus` borrows heavily from the `.litmus` format of the original litmus tool [21] and at each section, we describe the differences from the GPU and original `.litmus` format.

Line 1 starts the test with the name of the architecture and a test name (we dub the PTX architecture `GPU_PTX`) and the test is SB. Lines 2–11 make up the register declarations and initialization. As noted in Section 2.3, PTX has typed registers and as such, we require all registers to be declared in this section. The syntax for declaration is: {*tid*}`:.reg` {*type*} {*register*} where *tid* is an integer thread identifier (e.g. 0 for T0), *type* is a PTX type (listed in Table 2.2), and *register* is a string of the form `r`{n} where n is an integer from zero to nine. It is also required that registers requiring a non-zero initialization be initialized here.

```
1  GPU_PTX SB
2  {
3  0:.reg .s32 r0;
4  0:.reg .s32 r2;
5  0:.reg .b64 r1 = x;
6  0:.reg .b64 r3 = y;
7  1:.reg .s32 r0;
8  1:.reg .s32 r2;
9  1:.reg .b64 r1 = y;
10 1:.reg .b64 r3 = x;
11 }
12
13 T0                | T1                ;
14 mov.s32 r0,1      | mov.s32 r0,1      ;
15 st.cg.s32 [r1],r0 | st.cg.s32 [r1],r0 ;
16 ld.cg.s32 r2,[r3] | ld.cg.s32 r2,[r3] ;
17
18 ScopeTree
19 (device (cta (warp T0) (warp T1)))
20
21 y: global, x: shared
22
23 exists
24 (0:r2=0 /\ 1:r2=0)
```

Figure 3.2. Example of a GPU `.litmus` file which specifies the store buffering (SB) test

In lines 5 and 6, we initialize T0's registers `r1` and `r3` to the memory locations `x` and `y`, respectively. We initialize T1's registers `r1` and `r3` similarly in lines 9 and 10. The original `.litmus` format also has the initialization section, but does not require register declarations, because it deals exclusively with architectures which do not have typed registers.

Lines 13–16 describe the concurrent program to be run by the test. The concurrent program consists of several sequential programs (to be ran concurrently) given in vertical columns and separated with a pipe (`|`) character. Each sequential program starts with a thread identifier of the form `T`$\{n\}$ where n is a integer from zero to nine. Following the thread identifier is a sequence of PTX instructions. We support all instructions listed in Table 2.2 along with several other basic binary operations (e.g. `add`, `xor`). The original `.litmus` format specified programs in the same manner, but did not have a parser for PTX programs.

As discussed in Section 2.5.1, a GPU litmus test must specify the location of testing threads in the GPU thread hierarchy. Recent literature has referred to these hierarchy levels as *scopes* [29, 32]; we adhere to that terminology here and require each GPU `.litmus` file to contain a *scope tree* declaration which specifies the testing threads locations in the GPU thread hierarchy. Syntactically this declaration begins with the keyword `ScopeTree` followed by an S-expression [45] representing a tree of depth four where each level corresponds to a level in the GPU thread hierarchy. Each list begins with an identifier for the level of the hierarchy. From top to bottom, these identifiers are `device`, `cta`, and `warp`. A `warp` list is simply a list of testing thread IDs (e.g. `T0`, `T1`). In the concrete example shown in Figure 3.2, the scope tree is declared on lines 18 and 19; threads T0 and T1 are in the same CTA but different warps. More scope tree examples are shown in Figure 3.3. The original `.litmus` format did not test GPUs and hence had no need for a scope tree.

```
//scope tree for a 2 threaded test where T0 and T1 are in different ctas
(device (cta (warp T0)) (cta (warp T1)))

//scope tree for a 2 threaded test where T0 and T1 are in different
//warps, but the same CTA
(device (cta (warp T0) (warp T1)))

//more involved scope tree for a 3 threaded test where T0 and T1 are
//in different warps, but the same CTA but T2 is in a different CTA
(device (cta (warp T0) (warp T1)) (cta (warp T2))))
```

Figure 3.3. Additional examples of scope tree declarations

A GPU `.litmus` test must also specify in which region of memory testing locations are, shared or global. To this end, a *memory map* declaration that appears immediately following the scope tree declaration is required. The syntax is: {*loc*}:{*region*} where *loc* is a memory location and *region* is either `shared` or `global`. This specifies that location *loc* will be in the *region* memory region. All memory locations used in the test must be placed in a memory region in this style. In the concrete example shown in Figure 3.2, the memory map is given on line 21 and specifies that location `y` is in the global memory region and location `x` is in the shared memory region. The original `.litmus` format did not test GPUs and hence had no need for a memory map.

The GPU `.litmus` test ends with an assertion about the final state of registers or memory locations. Syntactically, this begins with the keyword `exists` followed by an assertion in parenthesis. Registers are referred to with the following syntax {*n*}:{*reg*} where *n* is a thread integer identifier (e.g. `0` for thread T0) and *reg* is the name of a register declared in the initialization portion of the test. Memory locations are simply referred to by their name (e.g. `x`). We use the characters `/\` to refer to the conjunction operator and the equality symbol (`=`) to refer to equality. In the concrete example shown in Figure 3.2, the assertion on lines 23 and 25 asks if it is possible for register `r2` in thread T0 to equal `0` and register `r2` in thread T1 to equal `0`. The original `.litmus` has the same syntax for specifying assertions.

3.2 GPU Program Skeleton

The GPU program produced by litmus can be split into two parts, the CPU code and the kernel (i.e. GPU) code. While the GPU litmus tool can produce either CUDA or OpenCL code, for ease of presentation, we show only CUDA code in this section. We begin our discussion with the CPU code. Figure 3.4 shows the main loop executed by the CPU in the form of high-level functions. From top to bottom (and noted with the line number), we step through this loop:

- *line 3*: This loop runs the GPU litmus test `ITERATIONS` times. The `ITERATIONS` value can be controlled with a command line argument to the GPU litmus tool.

- *line 4*: The `initialize_gpu` kernel initializes all global memory used in the test. Recall that GPU memory cannot explicitly be accessed on the CPU and must either be initialized with a special CUDA function or in a separate kernel; we chose the latter and launch the GPU with a single thread in a single CTA to initialize global values.

```
1  ...
2  //main CPU loop
3  for(int i = 0; i < ITERATIONS; i++) {
4    initialize_gpu<<<1,1>>>();
5    test_kernel<<<ctas,threads>>>(*device_results);
6    record_global<<<1,1>>>(*device_results);
7    cudaMemcpy(*cpu_results, *device_results, cudaMemcpyDeviceToHost);
8    record_results(cpu_results);
9    }
10 display_results();
11 ...
```

Figure 3.4. Testing loop of the CPU portion of the generated program

- *line 5*: The `test_kernel` kernel runs the concurrent PTX test specified by the GPU `.litmus` file. The final contents of registers are placed in the global memory array `device_results` so that they may be copied back. It is launched with a variable number of CTAs and threads which we discuss in Section 3.4.1.

- *line 6*: The `record_global` GPU kernel records any global memory locations needed for the GPU litmus test assertion by placing them in the global memory array `device_results` so that they may be copied back to the CPU. Similar to the initialization kernel, this kernel is launched with a single thread in a single CTA as only several locations will ever need to moved.

- *line 7*: Here the final contents of registers and memory (which were copied to the global memory array `device_results`) are copied back to the CPU with the built-in CUDA `cudaMemcpy` function.

- *line 9*: Next the results are recorded in a histogram and checked against the assertion in the GPU litmus test.

- *line 11*: After running the test `ITERATIONS` times, A histogram of results with an emphasis on the results that satisfied the assertion is displayed. An example of the output for a CPU test is seen in Figure 2.7.

Next we discuss the high-level implementation of the GPU kernel which runs the concurrent PTX program specified in the GPU `.litmus` file. This kernel was referred to as `test_kernel` in Figure 3.4. The high level code is shown in Figure 3.5 which we discuss for the rest of this section.

```
1  //Inside the kernel test_kernel
2  ...
3  if (tid == T0_tid && wid == T0_wid && cid == T0_cid) {
4     //Execute T0's test
5     ...
6     //Record T0's registers
7  }
8  else if (tid == T1_tid && wid == T1_wid && cid == T1_cid) {
9     //Execute T1's test
10    ...
11    //Record T1's registers
12 }
13 ...
```

Figure 3.5. The kernel code where GPU threads execute the tests specified in the GPU `.litmus` file.

Recall that GPU threads all execute the same kernel and in order for certain threads to execute distinct code, they must branch on a conditional related to their thread ID, warp ID, and CTA ID (`tid`, `wid`, and `cid`, respectively) as seen in lines 3 and 8. Once testing threads (e.g. T0, T1) are filtered into their respective conditional code, they execute their program that is specified in the GPU `.litmus` file. After the program is executed, the threads record the values of their registers into a global memory array that the CPU can copy and record.

The testing thread IDs, testing warp IDs, and testing CTA IDs (e.g. `T0_tid`, `T0_wid`, and `T0_cid`, respectively) are determined by the GPU litmus tool and set such that the scope tree in the GPU `.litmus` test is satisfied. For example, if the scope tree specifies that T0 and T1 are in different CTAs, then `T0_cid` and `T1_cid` are never equal. Conversely if T0 and T1 are in the same CTA, then `T0_cid` and `T1_cid` must always be equal.

3.3 Critical Incantations

The code presented in Section 3.2 is quite simple and, if executed as is, does not expose any weak behaviors for any GPU litmus tests we ran. Speaking candidly, we had a difficult time observing weak behaviors on GPUs; this project suffered several failed attempts over the course of two years before we found success. We were finally able to observe weak behaviors when we developed two *critical incantations*, called such because without at least one of these incantations present, we are unable to observe weak behaviors. We dub these two incantations *general bank conflicts* and *memory stress*.

3.3.1 General Bank Conflicts

Recall from Section 2.2.1 that concurrent memory accesses on GPU hardware are susceptible to bank conflicts due to poor memory accesses patterns within a warp. CUDA documentation states that when a bank conflict occurs, memory accesses are serialized. We are not concerned with the performance consequences of these memory access patterns; rather we use them to cause stress on the memory system which we (correctly) hypothesized could cause executions revealing weak memory behaviors. Official documentation only refers to bank conflicts applying to the shared memory region. However, we observe that this incantation works just as well for memory locations in the global memory region; as such, we refer to this incantation as *general bank conflicts*.

This incantation lets all threads in the testing thread's warp execute the testing threads program. While the extra threads in the warp execute the same instructions as the testing thread, they are provided with dummy addresses for each memory access instruction. These dummy addresses are computed to be one of the following:

- **Parallel**: All threads in the warp will access their own memory bank for this memory access instruction.
- **Broadcast**: All threads in the warp access the same memory location as the testing thread for this memory access instruction. Note that this option is only computed for read memory accesses; nontesting threads writing to testing locations would cause corrupt results.
- **Bank Conflict**: All threads in the warp will cause a bank conflict with the testing thread on this memory access instruction.

In order to test many different GPU access patterns for a given test, the access type (i.e. parallel, broadcast, or bank conflict) for each memory access instruction is randomized for each iteration of the GPU test.

The implementation of this incantation is largely in the testing kernel and happens as we filter testing threads into their testing code. Figure 3.6 shows a snippet of code implementing this general bank conflict for a specific testing thread T0 in the GPU testing kernel. We describe in detail this code snippet next.

- *Line 3*: Here the testing thread is filtered only by warp ID `wid` and CTA ID `cid`, thus the entire warp of the testing thread enters this code.

```
1  ...
2  //T0's entire warp executes test
3  if (wid == T0_wid && cid == T0_cid) {
4     //Assign unique address that potentially cause bank conflicts
5     bc_x = compute_address(access_type_x, T0_tid, x, tid);
6     bc_y = compute_address(access_type_y, T0_tid, y, tid);
7     ...
8     //Execute T0's test with new addresses
9     ...
10    if (tid == T0_tid) {
11       //Record T0's registers
12       ...
13    }
14 }
15 ...
```

Figure 3.6. Code snippet of the general bank conflict incantation implementation

- *Line 5-6*: The example shows two memory locations `x` and `y`. New addresses `bc_x` and `bc_y` are computed (`bc` stands for bank conflict) via the `compute_address` function which will return the original `x` and `y` location for the testing thread, but different addresses for other threads in the warp. The access type argument (i.e. `access_type_x` and `access_type_x`) indicate what type of access (i.e. bank conflict, parallel, broadcast) will happen for each address. They are randomized per iteration.

- *Line 8*: All threads in the warp execute T0's test using their newly computed addresses.

- *Line 10-13*: Only the testing thread (T0) records the results.

While this incantation is one of our key ingredients for observing weak behaviors on GPUs, it does carry some consequences. Specifically, there must now be enough continuous memory space starting at the testing locations to allow all 32 threads in the warp to cause bank conflicts with the testing thread. Given that bank conflicting addresses are 32 words apart, this requires $32 * 32 * 4 = 4096$ bytes of memory per testing location as opposed to simply four bytes before. Given the amount of memory on current GPUs (over 1 GB for global memory and 64 KB for shared memory), this is not an issue for tests with a small number of testing locations.

3.3.2 Memory Stress

Memory systems implemented on modern multiprocessors have complicated caching protocols which implement involved eviction and write-back policies [14]. Our hypothesis is that stressing this system with relentless memory accesses will put these protocols in interesting states and, in turn, trigger weak memory model executions. For example, a memory bus may be more likely to transfer data out of order when it is under heavy stress then if it is only servicing several requests. To this end, all nontesting threads are employed to read and write from nontesting memory locations for an incantation we dub *memory stress*.

We implement two functions, `mem_stress_write()` and `mem_stress_read()`, which repeatedly write and read to nontesting memory locations, respectively. These functions implement efficient GPU access patterns by ensuring accesses contain no bank conflicts and are largely optimally aligned; additionally, we make sure that warps do not diverge. This allows memory to be written to and read from as rapidly and by as many threads as possible.

The general bank conflict incantation described in Section 3.3.1 discusses that each testing memory location now uses 4096 bytes of memory where most of it is padding to allow for bank conflicts. Here, we take advantage of that padding memory, which is targeted by the memory stressing functions. We emphasize that these stressing functions do not touch the actual testing locations as that would interfere with the GPU litmus test. As a fail safe, the `mem_stress_write()` function writes chaotic values which would easily be recognizable as unwanted interference in the histogram of results.

The high-level code of how this incantation is implemented is shown in Figure 3.7. First, it is shown that testing threads are filtered off to perform the PTX program specified in the GPU `.litmus` file in lines 2–10. The remaining threads enter the memory stress region in lines 13–19. In our implementation, half of the warps (i.e. warps with even numbered warp ids) write to the memory, while the other warps read from memory.

We admit that there are many different ways to stress the memory system and due to the lack of intimate documentation about caching protocols implemented on these chips, we are unable to rigorously explain why these access patterns work as well as they do. However, we are able to observe that this technique is crucial for exposing weak behaviors and provide results which facilitate interesting observations and discussions about GPU memory models (e.g. see Chapter 4).

```
...
//Filter off T0
if (wid == T0_wid && cid == T0_cid) {
  ...
}
//Filter off T1
else if (wid == T1_wid && cid == T1_cid) {
  ...
}
...
//All threads not testing, stress the memory system
else {
  //Even number warps do the writes
  if (wid % 2 == 0)
     mem_stress_write();

  //Odd numbered warps do the reads
  else
     mem_stress_read();
}
...
```

Figure 3.7. High-level structure of the memory stress incantation implementation

3.4 Extra Incantations

In this section, we present two additional incantations that we call *random threads* and *synchronization*. While these incantations are not critical (i.e. weak behaviors are observed without them), their presence dramatically increases the number of weak memory behaviors we observe. We report on the effectiveness of these incantations in Section 3.5.

3.4.1 Random Threads

To test many different physical locations of testing threads on the GPU, the launch parameters (i.e. how many threads and CTAs with which the kernel is launched) and global IDs of testing threads are randomized for each iteration of the test. We call this incantation *random threads*. To implement this, global IDs (a combination of thread ID, warp ID, and CTA ID) are randomly assigned to testing threads such that the scope tree given in the specification remains valid. The memory model described in NVIDIA documentation [19, 20] (which is what we hope to test and eventually formalize) is unaware of concrete global IDs (e.g. thread ID = 1, CTA ID = 2); the model simply cares about the relationship between global IDs. That is, the model gives ordering guarantees based on if threads are interacting within the same CTA or across different CTAs. This incantation attempts to

get a good sampling of different concrete IDs over the relationships specified in the scope tree of the GPU `.litmus` file.

Randomizing global IDs and launch parameters can have several consequences for how the testing threads are executed on hardware. For example, multiple CTAs may be scheduled on an SM (streaming multiprocessor) if there is enough resources. By randomly selecting the number of threads per CTA (one of the limiting factors in how many CTAs are scheduled on an SM) and the CTA ID of testing threads, we allow the opportunity for testing threads to be mapped to a variety of SM assignments across the GPU. This may even allow some threads to be executed on the same SM, while others are on different SM.

Documentation states that when a bank conflict happens, memory accesses are serialized [19, p. 187]. Given that the general bank conflict incantation is a critical incantation, we believe that this serialization may facilitate weak memory model executions. For example, we hypothesize that two memory instructions may be reordered if one access is issued completely concurrently while the other must be completely serialized. By randomly assigning the thread ID of testing threads, our hope is that the testing thread is placed in a variety of places in the serialization order, thus exposing more weak behaviors.

3.4.2 Synchronization

To allow testing threads to execute their respective tests closely in sync with one another, and hence promote interactions while memory values are actively moving through the memory system, testing threads synchronize immediately before the PTX programs specified in the GPU `.litmus` file are executed. This incantation is borrowed directly from the original litmus tool and is called *synchronization*.

As a notable difference, GPUs do not guarantee forward-progress for interactions at certain levels of the GPU thread hierarchy, and naive synchronization implementations are prone to deadlock. Specifically, CTAs are not guaranteed to be scheduled concurrently [19, p. 12] and threads in the same warp do not have forward progress guarantees with respect to each other [46]. To ensure that CTAs will be scheduled concurrently, we adopt the *persistent thread model* presented in [47] in which the number of CTAs launched is limited to be *at most* the number of SMs on the GPU. Because each SM can run at least one CTA, this ensures all CTAs will be ran concurrently. To ensure threads within a warp do not deadlock, only a single thread per warp (i.e. the testing thread) is allowed to execute the synchronization barrier; this method was presented in [27]. Due to the warp synchronous execution model of the GPU, the other threads in the warp will not continue execution until the testing thread has been released from the barrier.

Because only one instance of the GPU litmus test is executed per kernel, the barrier implementation needs only to synchronize testing threads once. This is accomplished via an atomic add instruction and a spin loop. The barrier values are reset at each iteration in the `initialize` kernel called in the main CPU loop.

3.5 Effectiveness of Incantations

In this section, we discuss the effectiveness of the incantations described in Sections 3.3 and 3.4. We benchmark all combinations of critical and extra incantations by running several tests which attempt to expose different reorderings and find the most effective incantations for different GPU configurations (see Section 2.5.1 for the configurations we test). We run each test 100,000 times on three different GPU chips across three generations of architectures; from oldest to newest, these chips are Tesla C2075 (Fermi), GTX Titan (Kepler), and GTX 750 (Maxwell). We report the average the number of weak behaviors observed per set of incantations.

3.5.1 Inter-CTA Incantations

We first consider how effective incantations are for inter-CTA GPU configurations. We benchmark three tests, chosen for the different reorderings they attempt to expose. These tests are:

- **Message Passing (MP)**: This test is described in Section 5.2.1 and tests a handshake idiom.
- **Load Delaying (LD)**: This test (also known as load buffering) is described in Section 5.2.2 and tests if load operations may be reordered with program order later write operations.
- **Store Buffering (SB)**: This test is described in Section 5.2.3 and tests if store operations may be reordered with program order later read operations.

These benchmarks are provided to give a general idea of how effective incantations are and not as an exhaustive study on how to most effectively run individual tests. Therefore, we limit our benchmarking to these basic tests and do not consider tests with fences or other synchronization constructs (e.g. dependencies).

Table 3.1 shows the results of running these tests under different incantation combinations. The first column specifies the critical incantation used. Notice that if no critical incantation is present, no weak behaviors are observed despite the presence of

Table 3.1. Effectiveness of incantations for inter-CTA GPU configurations

Critical Incantations	Extra Incantations	MP	LD	SB
None	None	0	0	0
	Randomization	0	0	0
	Sync	0	0	0
	Randomization + Sync	0	0	0
General Bank Conflicts	None	836	0	0
	Randomization	1984	0	3
	Sync	0	0	0
	Randomization + Sync	2867	0	2
Memory Stress	None	234	653	760
	Randomization	290	313	291
	Sync	6614	211	268
	Randomization + Sync	4878	2838	3328
Memory Stress + General Bank Conflicts	None	73	28	6
	Randomization	368	92	93
	Sync	202	223	35
	Randomization + Sync	2901	636	716

extra incantations. For extra incantations, we write *randomization* for the random thread incantation discussed in Section 3.4.1 and *sync* for the synchronization incantation described in Section 3.4.2. We use the plus (+) symbol between two incantations when both are present.

We observe from the results in Table 3.1 that the number of weak behaviors observed is highly dependent on both the test and incantations used. For example, for MP with general bank conflicts, we are unable to observe weak behaviors with the only the sync extra incantation. We observe that the memory stress critical incantation with both sync and randomization seems to be the most effective set of incantations; however, LD and SB are greatly reduced if sync or randomization are used exclusively as the extra incantations.

3.5.2 Intra-CTA Incantations

We now consider how effective incantations are for intra-CTA GPU configurations. While in Section 3.5.1, we were able to use different tests to show the effectiveness of incantations, the only one of the three tests (MP, LB, SB) that we are able to observe for intra-CTA configurations is MP. This may be because for intra-CTA configurations, our incantations are still not enough to expose weak behaviors, or because there is a stronger memory model implemented at this level. Because we are only able to observe MP, we only show results for this test. We have two variants of the MP test at this GPU configuration which are:

- **Message Passing Global (MP-Global)**: This is the same message passing test used in Section 3.5.1, except that under this GPU configuration, all threads are in the same CTA and target the global memory region.

- **Message Passing Shared (MP-Shared)**: This is the same message passing test as MP-global, but in this GPU configuration, all memory accesses target the shared memory region.

Table 3.2 shows the results of running these tests under different incantation combinations. Similar to the inter-CTA tests, we observe that critical incantations are required for observing any weak behaviors and that the number of weak behaviors observed is highly dependent on both the test and incantations used. In the intra-CTA tests, the general bank conflict incantation is by far the most effective; in fact, the memory stress incantation by itself produces very few if any weak behaviors. This is the opposite of what we observed for the inter-CTA tests where memory stress was the most effective critical incantation. Additionally, for intra-CTA tests, the sync incantation without the randomization will produce no weak behaviors.

This section shows that the effectiveness of incantations depends heavily on the GPU configuration of the test. Currently, all incantations are controllable via command line

Table 3.2. Effectiveness of incantations for intra-CTA GPU configurations

Critical Incantations	Extra Incantations	MP-Global	MP-Shared
None	None	0	0
	Randomization	0	0
	Sync	0	0
	Randomization + Sync	0	0
General Bank Conflicts	None	877	0
	Randomization	2150	2061
	Sync	0	0
	Randomization + Sync	1989	2223
Memory Stress	None	7	0
	Randomization	7	0
	Sync	2	0
	Randomization + Sync	0	0
Memory Stress + General Bank Conflicts	None	0	0
	Randomization	336	1249
	Sync	0	0
	Randomization + Sync	1360	1722

arguments. Future work may analyze tests and dynamically configure incantations based on the GPU configuration in the test.

CHAPTER 4

NOTABLE RESULTS AND CASE STUDIES

In this chapter, we discuss notable testing results and case studies of CUDA applications. We go over some initial notations and considerations in Section 4.1. The first results that we discuss are interesting with respect to general memory consistency properties (e.g. coherence) and documentation in the PTX ISA manual [20]. Specifically, Section 4.2 shows that some deployed GPUs implement controversial relaxed coherence behaviors. Section 4.3 discusses the L1 cache memory annotation on Fermi architectures and how it cannot be used reliably for *any* inter-CTA interactions; this has programming consequences as it is the default memory annotation for the CUDA compiler. Section 4.4 tests the `.volatile` memory annotation and compares our observations with vendor documentation.

The second half of this chapter presents CUDA case studies where developers have made assumptions about the GPU memory model which may lead to erroneous behaviors. Section 4.5 discusses two GPU spin-locks which do not use fences: one from the popular *CUDA by Example* book [2] and the other from Owens and Stuart's paper entitled *Efficient Synchronization Primitives for GPUs* [48]. Both of these lock implementations assume that read-modify-write atomics provide sequentially consistent behavior; however, we show that this is not the case. We conclude by examining a GPU concurrent deque appearing in both a publication [49] and the book *GPU Computing Gems: Jade Edition* [50, pp. 485–499]. We show that the provided fence-less implementation could lead to the undesirable case of stale data being read from the deque.

4.1 Notations and Considerations

In the tests presented in this chapter, we use a parameterizable fence instruction that we note `membar.`{*scope*}. This fence is then instantiated for the different `membar` scopes, namely `.cta` and `.gl` (the third scope `.sys` is used only a few times in this document for reasons given in Section 5.1). We say that the `membar` has scope *None* for tests with no fence. Some tests have more than one fence instruction; however, in this chapter, we only

consider tests where both fences have the same scope annotation. That is, for scope `.cta` all `membar`s will have the `.cta` annotation. While this does not test all possible combination of fences, this chapter is largely concerned with testing *if* weak behaviors are observed, and if so, is it possible to experimentally disallow them. To that end, we do not enumerate all fence combinations. All testing results come from running 100,000 iterations.

Additionally, we observe far fewer weak behaviors on the GTX 750 (Maxwell) chip than the other chips. We hypothesize several reasons for this. The GTX 750 is a substantially smaller chip than the others (having only 5 SMs); this means there are less physical resources to run threads that stress the memory system in the crucial memory stress incantation (see Section 3.3.2). Another reason might be that we have not fine tuned our tool to test this chip, given that it has only been available for a few months at the time of writing. Finally, this chip may simply implement a stronger model than the others.

4.2 Coherence of Read-Read (CoRR)

Coherence is a property of memory consistency that applies only to single address behaviors. It has been defined as SC for a single address [14, p. 14]. Nearly all modern CPU memory models guarantee coherence, with the exception of Sparc RMO [51, pp. 265–267] which allows reads from the same address to be reordered. This behavior can be seen in the coherence of read-read (or CoRR) litmus test; a PTX instance of this test is shown in Figure 4.1. In this test, T1 is able to read the updated value from memory followed in program order by a read which returns stale data. If this behavior is allowed, we would like to investigate which memory fence (i.e. `membar`) placed in between the loads in T1 is required to disallow it.

This weak behavior (i.e. CoRR) has been controversial in CPU memory models as it is observable on many ARM chips but confirmed as buggy behavior [16, 52]. Additionally, new language level memory models (e.g. OpenCL 2.0 [53] and C++11 [9]) disallow this behavior and it is unclear how to efficiently implement such languages on hardware with

initial state: x = 0	
T0	T1

`st.cg.s32 [x], 1 ;`	`ld.cg.s32 r1, [x] ;`
	`membar.{`*`scope`*`}  ;`
	`ld.cg.s32 r2, [x] ;`

assert: 1:r1=1 ∧ 1:r2=0	

Figure 4.1. Test specification for CoRR

this relaxation. We test this behavior on GPUs and show that older architectures (Fermi and Kepler) allow this behavior, but newer chips (Maxwell) experimentally do not.

Table 4.1 shows the results of running the CoRR test on three GPUs with all different architectures (Fermi, Kepler, and Maxwell). We test all three GPU configurations described in Section 2.5.1. We observe that CoRR is indeed observable on Kepler and Fermi architectures for all GPU configurations but is not observable at all on the newer Maxwell architecture. We observe that only the smallest scoped fence `membar.cta` is required to experimentally disallow this test for any of the tested GPU configurations.

4.3 Fermi Memory Annotations

Recall that the `.ca` memory annotation loads from the L1 cache (see Table 2.2) and that separate CTAs may have separate L1 caches if they are mapped to different SMs (see Section 2.2). The PTX manual [20, p. 121] explicitly states that multiple L1 caches are incoherent by stating:

> Global data is coherent at the L2 level, but multiple L1 caches are not coherent for global data. If one thread stores to global memory via one L1 cache, and a second thread loads that address via a second L1 cache with `ld.ca`, the second thread may get stale L1 cache data, rather than the data stored by the first thread.

In this section, we test the L1 memory annotation (i.e. `.ca`) across CTAs to determine what extent this operator can be used reliably for inter-CTA interactions.

4.3.1 Message Passing Through L1

Consider the test shown in Figure 4.2. This type of test is named message passing (MP) and describes a handshake idiom. Specifically, T0 writes some data to location `x` followed

Table 4.1. Results for CoRR tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-warp:S-cta-Shared	*None*	7356	8572	0
	`.cta`	0	0	0
	`.gl`	0	0	0
D-cta:S-ker-Global	*None*	3668	10047	0
	`.cta`	0	0	0
	`.gl`	0	0	0
D-cta:S-ker-Global	*None*	3246	4769	0
	`.cta`	0	0	0
	`.gl`	0	0	0

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`ld.ca.s32 r1, [y] ;`
`membar.{scope} ;`	`membar.{scope} ;`
`st.cg.s32 [y], 1 ;`	`ld.ca.s32 r2, [x] ;`
assert: 1:r1=1 ∧ 1:r2=0	

Figure 4.2. Test specification for MP-L1

by a flag to location `y`. We test if T1 is allowed to read the updated flag followed by a read in program order that returns stale data. Notice that we use the `.ca` memory annotation for all load operations; we dub this test MP-L1. Because our aim here is to test multiple L1 caches, we only consider the GPU configuration where T0 and T1 are in different CTAs and thus, `x` and `y` must be in the global memory region. This corresponds to the GPU configuration D-cta:S-ker-Global.

We report the results of running MP-L1 in Table 4.2. We observe that on Fermi architectures, *no fence is strong enough to disallow the MP-L1 test*. To emphasize this point, we include the `.sys` fence in our tests which we largely exclude from this document for reasons explained in Section 5.1. We emphasize that the `.sys` is documented to be the strongest fence in the PTX documentation, as it enforces orderings across all interactions including multidevice interactions [20, p. 169]. We observe that not even the `.sys` fence restores orderings to this example on Fermi architecture; however, this behavior is able to be experimentally disallowed on Kepler and Maxwell with what we interpret to be the appropriately scoped fence (i.e. `membar.gl`). This behavior not appearing on Kepler and Maxwell is possibly because the documentation states that the L1 cache has been disabled for global memory accesses on these architectures and global memory is cached in the L2 cache regardless of the memory annotation [19, p. 194]. That is, we believe this issue to be

Table 4.2. Results for MP-L1 tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-cta:S-ker-Global	*None*	11648	8129	3
	`.cta`	455	3087	0
	`.gl`	208	0	0
	`.sys`	201	0	0

the result of multiple L1 caches interacting; if the L1 cache is disabled for global memory accesses, then we will not see the symptoms of their interactions.

4.3.2 Mixing Memory Annotations

The previous section showed that inter-CTA interactions cannot implement a message passing (MP, or handshaking) protocol reliably (i.e. disallow stale values from being read from memory) using the `.ca` exclusively for loads. In this section, we mix memory annotations in an attempt to restore orderings between multiple L1 caches. We hypothesize that perhaps we may be able to propagate values up from the L2 cache to the L1 cache by reading the cache line first from the L2. We get this hypothesis from the PTX ISA manual which states that after an L2 load (i.e. `.cg`), "... existing cache lines that match the requested address in L1 will be evicted" [20, p. 121]. While it is not clear what guarantees (if any) are provided in this quote, it seems to suggest that a read from the L2 will somehow effect the L1 cache (e.g. by evicting values).

The most basic test we could think of to examine this behavior is a variation of CoRR (see Section 4.2) where we first read data from the L2 cache via the `.cg` memory annotation and then attempt to read the same data from the L1 cache via the `.ca` annotation. This would correspond to the memory value being propagated up the cache hierarchy (from the L2 to L1) after it is first read from the L2. This test, which we dub CoRR-L2-L1, can be seen in Figure 4.3.

The results of running this test are shown in Table 4.3. We observe that in the Fermi architecture, no fence is strong enough to guarantee that updated values will be read reliably from the L1 cache *even when they are first read from a shared cache.* Similar to Section 4.3.1, to emphasize this point, we include the `.sys` fence in our tests which we largely exclude from this document for reasons explained in Section 5.1.

initial state: x = 0	
T0	T1

`st.cg.s32 [x], 1 ;`	`ld.cg.s32 r1, [x] ;`
	`membar.{`*`scope`*`}  ;`
	`ld.ca.s32 r2, [x] ;`

assert: 1:r1=1 ∧ 1:r2=0	

Figure 4.3. Test specification for CoRR-L2-L1

Table 4.3. Results for CoRR-L2-L1 tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-cta:S-ker-Global	*None*	10247	4739	0
	`.cta`	1989	0	0
	`.gl`	1669	0	0
	`.sys`	1706	0	0

4.3.3 CUDA Programming Consequences

Because of the two previous results, we are convinced that on Fermi architectures, the `.ca` memory annotation *cannot be used for reliable inter-CTA communication at all* (i.e. it is not possible to disallow stale values from being read from memory). Interestingly, the `.ca` memory annotation is the default annotation for the CUDA compiler [20, p. 121]. Therefore, any programmer who wishes to develop GPU code with inter-CTA interactions needs to explicitly specify that the L2 memory annotation (i.e. `.cg`) be used. This can be accomplished with the `nvcc` command line argument: `-Xptxas -dlcm=cg -Xptxas -dscm=cg`. We show throughout Chapter 5 that we are able to reliably use fences to disallow stale values from being read when the L2 memory annotation is used.

As a further consequence, the (single) memory consistency example provided in the CUDA manual [19, p. 95] computes a reduction (i.e. summing the values of a vector) and uses a memory load to retrieve values across CTAs. Even though the example provides a fence, we have shown in this section that no fence is sufficient under default compilation schemes (i.e. `.ca` memory annotations) to disallow stale values from being read. Thus this example is broken on Fermi architectures if compiled without explicitly specifying the `.cg` annotation to be used, of which the CUDA guide makes no mention.

4.4 Volatile Operators

The PTX ISA provides the `.volatile` memory annotation with the following documentation [20, p. 136]: "`st.volatile` may be used with `.global` and `.shared` spaces to inhibit optimization of references to volatile memory. This may be used, for example, to enforce sequential consistency between threads accessing shared memory".

It is not clear to us which GPU configurations (i.e. inter or intra CTA and memory regions) to which this documentation is extending sequential consistency guarantees (or if fences are additionally required to provide sequential consistency); we see this phrasing as a potential source of confusion and test the behavior of this annotation in this section.

Figure 4.4 presents a simple MP style test using the `.volatile` annotation which we dub MP-volatile.

Table 4.4 shows the results of running this test on all GPU configurations discussed in Section 2.5.1. We observe that without fences, the `.volatile` annotation *does not enforce sequentially consistent behavior at any GPU configuration.* However, weak behaviors can be experimentally disallowed when (what we interpret to be) the appropriate fences are included (`.cta` or `.gl` for intra-CTA configurations and `.gl` for the inter-CTA configuration). While the exact intention of the documentation is unknown, we suggest a rewording to alleviate potential confusion. Tentatively, we suggest amending the original documentation as such:

> `st.volatile` may be used with `.global` and `.shared` spaces to inhibit optimization of references to volatile memory. This may be used in conjunction with the appropriate memory fence to enforce sequentially consistent executions between threads.

4.5 Spin-Locks

In this section, we test two GPU spin-lock mutex implementations; the first is given in the book *CUDA by Example* [2], the second is given by Jeff Stuart and John Owens in their paper *Efficient Synchronization Primitives for GPUs* [48]. We show that these implementions do not satisfy what is generally considered to be the correct specification for a mutex. Specifically, we show that a critical section may read data values that are stale w.r.t. the previous critical section for inter-CTA interactions. We then show that the addition of memory fences experimentally provides the expected behavior. We document these behaviors in terms of short litmus tests and the results of running them in our testing framework.

initial state: x = 0, y = 0	
T0	T1
`st.volatile.s22 [x],1 ;`	`ld.volatile.s32 r0, [y] ;`
`membar.{scope} ;`	`membar.{scope} ;`
`st.volatile.s32 [y],1 ;`	`ld.volatile.s32 r2, [x] ;`
assert: 1:r0=1 $\wedge$ 1:r2=0	

Figure 4.4. Test specification for MP-volatile

Table 4.4. Results for MP-volatile tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-warp:S-cta-Shared	*None*	2007	3078	0
	`.cta`	0	0	0
	`.gl`	0	0	0
D-cta:S-ker-Global	*None*	822	3025	0
	`.cta`	0	0	0
	`.gl`	0	0	0
D-cta:S-ker-Global	*None*	699	7948	7
	`.cta`	219	3120	0
	`.gl`	0	0	0

4.5.1 CUDA by Example

CUDA by Example presents a mutex implementation for combining CTA-local partial sums [2, pp. 251–254]. The mutex implementation is a simple atomic compare-and-swap (i.e. CAS) spin-lock with an atomic exchange release. We reproduce a simplified version of the lock and unlock functions in Figure 4.5 for reference. Note that the original implementation had an error which we have repaired as given in the official errata for the book (see `https://developer.nvidia.com/cuda-example-errata-page`).

The locks are used to update a global value `c` with the CTA-local partial sums located in `cacheIndex[0]`. Only one thread per CTA executes this code. This part of the implementation is shown in Figure 4.6.

While the book does not explicitly mention memory consistency issues, the following paragraph suggests that the behavior typically expected from a lock can be obtained by only using atomic operations. For context, it is explaining why `unlock` must be an atomic exchange rather than simply a store [2, p. 254].

```
1  __device__ int mutex;
2
3  __device__ void lock( void ) {
4    while( atomicCAS( mutex, 0, 1 ) != 0 );
5  }
6
7   __device__ void unlock( void ) {
8    atomicExch( mutex, 0 );
9  }
```

Figure 4.5. Implementation of lock and unlock given in *CUDA by Example*

```
1  ...
2  //cacheIndex is equal to tid
3  if (cacheIndex == 0) {
4   lock.lock();
5   *c += cache[0];
6   lock.unlock();
7   }
```

Figure 4.6. Code snippet from the mutex example given in *CUDA by Example*

> Atomic transactions and generic global memory operations follow different paths through the GPU. Using both atomics and standard global memory operations could therefore lead to an `unlock()` seeming out of sync with a subsequent attempt to `lock()` the mutex. The behavior would still be functionally correct, but to ensure consistently intuitive behavior from the application's perspective, it's best to use the same pathway for all accesses to the mutex.

We distill this mutex implementation into a GPU litmus test named CAS spin-lock (abbreviated to CAS-SL) shown in Figure 4.7. The reader may wish to refer back to Table 2.2 for a description of some of the PTX instructions used in this test. This test describes two threads interacting via a CAS spin-lock. The `y` memory location is the mutex and `x` is the global data accessed in the critical section. The test begins in a state where T0 has the mutex (`y` = 1). T0 stores a value to `x` and then releases the mutex with an atomic exchange. T1 attempts to acquire the lock with a CAS instruction, then checks if the lock was acquired successfully via the `setp` command. If the lock was acquired, i.e. `r0 = 0`, then T1 attempts to read the global data in `x`. This is enforced using PTX predicated execution [20, p. 160]; that is, instructions annotated with `@r1` will only execute if the `setp` command was satisfied. The final constraint describes an execution where T1 successfully acquires the lock (i.e. `1:r0 = 0`) yet does not see the updated value in `x` (i.e. `1:r2 = 0`).

Table 4.5 shows the test outcomes for variants of the CAS-SL test for three different chips. We only test GPU configuration D-cta:S-ker-Global because that is the interaction

initial state: x = 0, y = 1	
T0	T1
`st.cg.u32 [x], 1 ;`	`atom.cas.b32 r0,[y],0,1 ;`
`membar.{scope} ;`	`setp.eq.u32 r1, r0, 0 ;`
`atom.exch.b32 r0,[y],0 ;`	`@r1 membar.{scope} ;`
	`@r1 ld.cg.u32 r2,[x] ;`
assert: 1:r0=0 ∧ 1:r2=0	

Figure 4.7. Test specification for CAS-SL

Table 4.5. Results for CAS-SL tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-cta:S-ker-Global	*None*	86	1607	0
	`.cta`	17	692	0
	`.gl`	0	0	0

that is described in the *CUDA by Example* application (it is an inter-CTA mutex). We observe that without fences, T1 can indeed load stale values. While the `.cta` fence scope reduces the number of times we observe the weak behavior, the (`.gl`) fence is required to completely disallow the behavior based on our experimental results.

The CAS-SL test distills the locking behavior in *CUDA by Example* to a simple message passing idiom. If T1 is able to see a stale value, then the total sum could be computed without considering T0's contribution; this will lead to an incorrect summation result. The implementation in *CUDA by Example* has inter-CTA interactions and is lacking fence instructions which leaves the code vulnerable to this error.

4.5.2 Efficient Synchronization Primitives for GPUs

In their paper *Efficient Synchronization Primitives for GPUs*, Stuart and Owens provide synchronization primitives for GPUs [48]. They include a spin-lock that is similar to the one presented in Section 4.5.1, with the difference being that they use atomic exchange instead of compare-and-swap for the locking function. They continue to discuss how to optimize the mutex functions by reducing contention for a memory location using a method they refer to as a *backoff* strategy, which does not introduce any additional memory ordering operations (e.g. memory fences). The authors explicitly make the assumption that an atomic exchange can be used in place of a store and memory fence by stating [48, p. 3]: "Also, we use `atomicExch()` instead of a volatile store and `threadfence()` because the atomic queue has predictable behavior, `threadfence()` does not (i.e. it can vary greatly in execution time if other memory operations are pending)".

We were unable to locate unambiguous justifications for the above assumptions in any NVIDIA documentation (CUDA or PTX). The following paragraph from the PTX ISA may be related, but seems to be restricted to atomicity and single address interactions; it does not seem to account for memory accesses inside the critical section [20, pp. 166–167]:

> Atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer's responsibility to guarantee correctness of programs that use shared memory

atomic instructions, e.g., by inserting barriers between normal stores and atomic operations to a common address, or by using `atom.exch` to store to locations accessed by other atomic operations.

We distill this mutex implementation to a litmus test named exchange spin-lock (abbreviated to EXCH-SL) shown in Figure 4.8 which describes two threads interacting via an atomic exchange spin-lock. The description is identical to the CAS-SL test described in Section 4.5.1, except here atomic exchange is used for the locking mechanism instead of atomic compare-and-swap. The final constraint describes an execution where T1 successfully acquires the lock (`1:r0 = 0`), yet does not see the updated value in `x` (`1:r2 = 0`).

Table 4.6 shows the test outcomes for variants of the CAS-SL test for three different chips. We only test GPU configuration D-cta:S-ker-Global because that is the interaction that is described in the paper. We observe that without fences, T1 can indeed load stale values. The `.cta` fence reduces the number of times we observe the weak behavior; however, the (`.gl`) fence is required to disallow the behavior based on our experimental results.

While the paper *Efficient Synchronization Primitives for GPUs* does not provide concrete examples using the locking mechanisms, this test distills a simple locking message passing idiom one might implement using this mutex. Traditionally, lock implementations have provided sufficient synchronization to ensure that critical sections observe the most recent values computed in previous critical sections [14, p. 64]; that is, values protected by locks should have sequentially consistent behavior (sequential consistency is described

initial state: x = 0, y = 1

T0	T1
`st.cg.u32 [x], 1 ;`	`atom.exch.b32 r0,[y],1 ;`
`membar.{`*`scope`*`} ;`	`setp.eq.u32 r1, r0, 0 ;`
`atom.exch.b32 r0,[y],0 ;`	`@r1 membar.{`*`scope`*`} ;`
	`@r1 ld.cg.u32 r2,[x] ;`

assert: 1:r0=0 ∧ 1:r2=0

Figure 4.8. Test specification for EXCH-SL

Table 4.6. Results for EXCH-SL tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-cta:S-ker-Global	*None*	98	1468	0
	`.cta`	12	638	0
	`.gl`	0	0	0

in Section 2.4). As seen in our results in Table 4.6, this is not the case without fences. Although the paper makes no claims about formal synchronization properties, given the traditional properties of locks, we feel that it may not have been intentional to allow such behaviors.

4.6 Dynamic Work Balancing

In this section, we test certain behaviors of a concurrent deque used in dynamic load balancing on GPUs as seen in [49] and again presented in the book *GPU Computing Gems Jade Edition (Applications of GPU Computing Series)* [50, pp. 485–499]. This technique is used in two applications: octree partitioning and four-in-a-row game simulation. We show that the provided implementations allow threads to read partial or stale data from the work deque in certain situations due to weak memory orderings on the hardware. We could not find any mention of weak memory model considerations in either publication documenting these concurrent deques.

Specifically, the dynamic load balancing is set up in the form of work-stealing deques (one per CTA) containing abstract task types. We show that in the case when one thread steals a task immediately after it was pushed by another thread, the stealing thread may not observe the recently pushed task, yet the deque will be updated as if the recently pushed task was correctly stolen. This can lead to several undesirable situations, including skipping tasks or loading partial tasks if tasks are large enough to be split into several load instructions (as is the case in both provided applications).

While the provided implementations of the octree partitioning and four-in-a-row simulator are advertised as being for architectures with compute capability `sm_13`, our testing framework largely targets generic address instructions which are not supported earlier than `sm_20`. As such, we are unable to test on the advertised architecture. However, we believe this remains a substantial issue given that memory fences are supported on all architectures and the book *GPU Computing Gems Jade Edition* is used as a reference for modern GPU computing.

4.6.1 CTA Level Work Stealing Deques

Here we briefly describe the dynamic load balancing technique. This is achieved through concurrent work-stealing deques, one per CTA. The particular concurrent deque, described in [54], avoids expensive read-modify-write instructions in the common case.

In this deque, there is a separate global head and tail index value; tasks are added and removed by the deque owner from the tail index (leaving the head pointer on the opposite

side of the deque, see Figure 4.9). The tail points to an empty cell and is decremented to find a task.

If there are no tasks remaining in the deque, the CTA may try to steal a task from another CTA's deque at the head index. Because the deque owner and thieves are accessing the deque from different ends, expensive synchronization is not needed when the deque contains more than 1 element. Synchronization may be required between multiple thieves accessing the same deque, but stealing is claimed to be the less common case.

4.6.2 Synchronization Between Owner and Thief

We now describe in detail the interaction when an owner pushes a task and a thief immediately steals the task. We first reproduce the code for the `push` and `steal` functions (adapted from [50, pp. 485–499]) in Figure 4.10. Note that `head` is a structure that contains an index and a counter. The counter is provided to avoid the ABA problem [55], which does not arise in our simple interaction.

The purpose of this discussion is not to examine all possible interactions between a deque owner and a thief, but rather to examine and then test one particular interaction. This interaction starts with an empty deque (in this simplified interaction, all tasks are simply integers and locations are initialized to 0). This means that the `head` and `tail` indexes point at the same location as seen in Figure 4.11.

The deque owner then pushes a task to the deque (say the integer value `1`) via the `push` function presented above. Now `head` points to the value 1 and `tail` has been incremented as seen in Figure 4.12

At this point, another CTA attempts to steal from the deque by calling the `steal` function. First, it checks for an empty deque. As we can see, the deque is not empty. Next, the task pointed to by `head` (copied into the value `oldHead`) at location `0` is loaded which is the value `1`. Finally, the thief checks if another thief has already stolen the task using

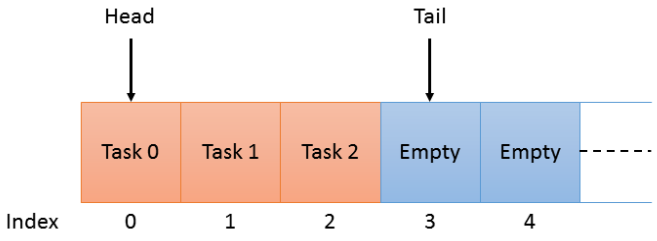

Figure 4.9. Example configuration of the concurrent deque

```
1  /* only 1 thread ever calls this function, therefore
2     no RMW required (e.g. atomic add) */
3  __device__ void push( task ) {
4    tasks[tail] = task;
5    tail++;
6  }
7
8  //steal function
9  __device__ Task steal( void ) {
10   int oldHead = head;
11
12   /* Check for empty deque */
13   if (tail <= oldHead.index)
14      return EMPTY;
15
16   task = tasks[oldHead.index];
17   newHead = oldHead;
18   newHead.index++;
19   if (CAS(&head, oldHead, newHead))
20     return task;
21
22   /* Unable to steal because of another thief */
23   return FAILED;
24 }
```

Figure 4.10. Implementation of push and steal for the concurrent deque

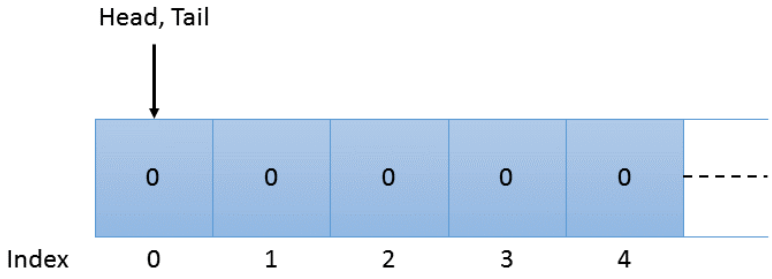

Figure 4.11. Initial state of the concurrent deque

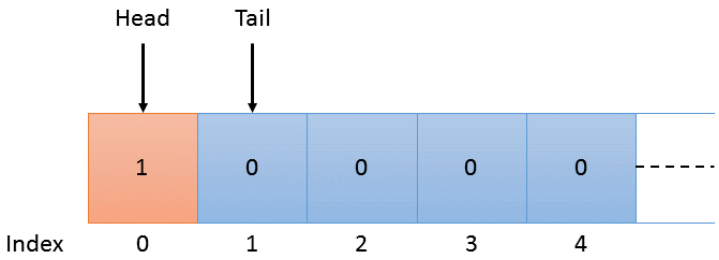

Figure 4.12. Concurrent deque after a single task has been pushed

the compare-and-swap function. We assume no other thief in this interaction, thus the task which contains `1` is returned and the head pointer is updated.

The above description is a correct execution of the deque. However, if the thief observed a nonempty deque and updated the head pointer, yet returned a stale value (i.e. `0`), we believe that would be a erroneous execution. This would correspond to the task that was just pushed by the owner being skipped, as the deque is updated to believe that task was correctly stolen, yet the thief has a stale task.

4.6.3 Test Distillation

We now distill this behavior into a simple litmus test that we can run using our tools. We call this test Dynamic Load Balancing Message-Passing (which we abbreviate DLB-MP) because it has a message passing style where T0 stores two values and we test if T1 may read stale values. Its specification is given in Figure 4.13.

Here we describe the DLB-MP test. First, there are two global variables `t` and `d` where `t` is the tail variable and `d` is the memory location at index 0 being pushed to and stolen from. Both are initialized to 0.

initial state: t = 0, d = 0	
T0	T1
`st.cg.u32 [d], 1 ;`	`ld.volatile.u32 r0, [t] ;`
`membar.{scope} ;`	`setp.eq.u32 r1, r0, 0 ;`
`ld.volatile.u32 r2, [t] ;`	`@r1 membar.{scope} ;`
`add.s32 r2, r2, 1 ;`	`@r1 ld.cg.u32 r2,[d] ;`
`st.volatile.u32 [t], r2 ;`	
assert: 1:r0=1 ∧ 1:r2=0	

Figure 4.13. Test specification for DLB-MP

T0 is the deque owner and T1 is the thief. Via the `push` function, T0 stores the task `1` to `d`. The provided implementations declare `tail` as `volatile`, which means that `tail++` will be compiled to a `volatile` load, followed by an increment, and a `volatile` store (as seen in T0's program).

T1 captures only the instructions of the thief that access tasks in the array and the tail index. We do not include accesses of the head index and instead provide the concrete value of 0 in the distilled test. We include the read of tail in line 5 of `steal` (the conditional on line 13 in Figure 4.10) as the first instruction in T1. Recall that `tail` is volatile which means the access will go to memory with the `.volatile` annotation. Next T1 includes a test of the value read from tail, corresponding to the emptiness check, again on line 5 (the conditional on line 13 in Figure 4.10). The last instruction in T1 is a conditional load of `d`, dependent on the outcome of the check. That is, we only load from the array if we saw the owner increment `t`, meaning that the deque is not empty. The thief would then return the result of loading from the array as a task while updating the head pointer.

We check the condition that if the thief sees the incremented `t` value (i.e. `1:r0=1`) and then loads the task from `d`, the task should *not* be the value 0. An execution satisfying these conditions would mean that the thief loaded a stale value, yet updated the deque to a state in which the correct task was stolen. As a consequence, the task pushed by the owner would be skipped or only partially loaded.

We have hand checked that our PTX test is similar to the PTX code generated from compiling the provided implementation in terms of memory access types and conditionals; we provide a snippet of the relevant PTX code compiled from the application in Appendix A.

4.6.4 Test Results

Table 4.7 shows the test outcomes for variants of the DLB-MP test for three different chips. We only test GPU configuration D-cta:S-ker-Global because that is the interaction described in the dynamic load balancing applications. We observe that without fences, T1 can indeed load stale values. While the `.cta` fence reduces the number of times we

Table 4.7. Results for DLB-MP tests

GPU Configuration	Fence	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
D-cta:S-ker-Global	*None*	5	750	0
	`.cta`	0	51	0
	`.gl`	0	0	0

observe the weak behavior, the (`.gl`) fence is required to disallow the behavior based on our experimental results.

Completely repairing concurrent data structures on architectures with weak memory models is notoriously difficult, especially when the vendor documentation is sparse and unclear (as is often the case). As such, a fix to the entire load balancing deque is outside the scope of this document; we simply show that `membar.gl` on each thread will experimentally outlaw the specific execution described above.

CHAPTER 5

BULK RESULTS

In this chapter, we report the results of running a wide range of tests on GPUs with different synchronization constructs. We first describe our notations for tests in Section 5.1, which closely resembles the notations used in [16]. Using these notations, we report on running large families of tests (i.e. base tests plus variants with different synchronization attributes) in Section 5.2 and make observations about the tests in Section 5.3. We then use these results to invalidate a previously proposed GPU model [31] in Section 5.4. The tests ran in this section were generated by Daniel Poetzl's GPU extensions to the DIY test generation tool [43].

5.1 Naming and Synchronization

To avoid having to give a code listing for all of the variants of tests in this document, we employ a naming convention that indicates attributes of the tests. First, test names are assigned to simple tests (e. g. MP is short for message passing). Then, *synchronization attributes* are indicated after a plus (+) sign. A table of synchronization attributes we consider can be found in Table 5.1. For example, the base MP test (shown earlier in Figure 2.8) can be modified to include a `membar.cta` in between the two memory accesses on T0 and a `membar.gl` in between the two memory accesses in T1. This modification yields the new test we call MP+membar.cta+membar.gl (shown in Figure 5.1).

Table 5.1. Test attributes

Attribute	**Description**
membar.cta	intra-CTA fence
membar.gl	intradevice fence
addr	address dependency
data	data dependency

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`ld.cg.s32 r1, [y] ;`
`membar.cta ;`	`membar.gl ;`
`st.cg.s32 [y], 1 ;`	`ld.cg.s32 r4, [x] ;`
assert: 1:r1=1 ∧ 1:r4=0	

Figure 5.1. Test specification for MP+membar.cta+membar.gl

We use the convention that names like MP+membar.cta+membar.cta are shortened to MP+membar.ctas (shown in Figure 5.2). Finally, we use the notation +* to refer to all possible variants.

The base test names are often given as an acronym describing the behavior they test. For example, SB stands for store buffering, and the SB test (Section 5.2.3) checks for store buffering behavior. However, some tests do not have such simple descriptions and intuitive acronyms have not yet been developed. For these tests, we simply use the names that have been used in past literature (e.g. for the POWER memory model in [16]). These tests are: 2+2W, R, and S in Sections 5.2.5, 5.2.6, and 5.2.7, respectively. To alleviate confusion, every base test is presented with an informative title (possibly separate from its name), a test specification, and a brief description.

5.1.1 Different Kinds of Synchronization

We consider the following PTX fences for synchronization:

- **membar.cta**: The PTX ISA manual [20, p. 165] describes this barrier as follows: "Waits until all prior memory writes are visible to other threads in the same CTA. Waits until prior memory reads have been performed with respect to other threads in the CTA".

Initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`ld.cg.s32 r1, [y] ;`
`membar.cta ;`	`membar.cta ;`
`st.cg.s32 [y], 1 ;`	`ld.cg.s32 r4, [x] ;`
assert: 1:r1=1 ∧ 1:r4=0	

Figure 5.2. Test specification for MP+membar.ctas

- **membar.gl**: The PTX ISA manual [20, p. 165] describes this barrier as follows: "Waits until all prior memory requests have been performed with respect to all other threads in the GPU. For communication between threads in different CTAs or even different SMs, this is the appropriate level of membar".

The PTX ISA manual describes a third barrier, `membar.sys`, as being appropriate for interdevice interactions. Because we have not yet tested interdevice behaviors and have not seen any counterintuitive behavior with this fence, we omit our `membar.sys` results for most of this document, including this entire chapter. We use this fence for two tests in Section 4.3 simply to show that no available fence can restore orderings with certain memory annotations.

Dependencies from load operations to program order later load or store operations have been given ordering properties in CPU memory models (e.g. the POWER model in [16]). In this document, we explore two types of dependencies:

- **Data Dependency**: This dependency is between a load and a store if there is a data-flow path through registers and instructions from the value loaded to the value written.

- **Address Dependency**: This dependency is between a load and a load or store if there is a data-flow path through registers and instructions from the value loaded to the address of the load/store.

An example of the load delaying (LD) test with a data dependency in both threads (referred to as LD+datas) is shown in Figure 5.3. Address dependency tests are computed similarly. Implementation for control dependencies (i.e. a dependency through a conditional branch) have not been implemented and we leave for future work.

We have observed the PTX assembler removing some of the trivial dependencies. For example, in Figure 5.3, the PTX assembler recognizes that a value XOR'ed with itself is

initial state: x = 0, y = 0	
T0	T1
`ld.cg.s32 r0, [x] ;`	`ld.cg.s32 r0, [y] ;`
`xor.b32 r1, r0, r0 ;`	`xor.b32 r1, r0, r0 ;`
`add.s32 r2, r1, 1 ;`	`add.s32 r2, r1, 1 ;`
`st.cg.s32 [y], r2 ;`	`st.cg.s32 [x], r2 ;`
assert: 0:r0=1 ∧ 1:r0=1	

Figure 5.3. Test specification for LD+datas

0 and will remove the dependency. Daniel Poetzl has developed a technique using higher order bits in some values to ensure that the dependency will not be optimized out. Because our tests use only small values (e.g. `0`, `1`), we are able to do simple bit-wise operations on the higher order bits of these values which result in 0; yet because the compiler does not know the range of values we use, it will not remove the dependency. We have checked via the `cuobjdump` output that the dependencies are still present in the machine-level assembly.

Furthermore, all memory access instructions use the same L2 memory annotation (`.cg`). This is because results presented in Section 4.3 show that fences cannot prevent weak behaviors when using the default L1 memory annotation (`.ca`). One of the goals of this section is to test which fences disallow weak behaviors; this should aid developers in knowing what synchronization (if any) is experimentally required to restore enough ordering for their programs to be correct. Given this, operations in which order cannot be enforced with synchronization are of little interest.

5.2 Test Specifications and Results

In the following sections, we describe families of tests which are generated from cycles described in [16] and their outcomes. Each test section contains a test specification of the base test and a brief description. A summary of the results is shown in a table that is split into three sections, one for each GPU configuration discussed in Section 2.5.1. For each test, the table states how often the tested behavior (specified by the final condition) occurred when the test was executed 100 000 times. We test three GPU chips over three generations. From oldest to newest, these chips are a Tesla C2075 (Fermi), a GTX Titan (Kepler), and a GTX 750 (Maxwell). For readability, our results label these chips after their respective generation; for example, Fermi refers to the Tesla C2075 chip, Kepler refers to the GTX Titan chip, and Maxwell refers to the GTX 750 chip.

Naturally, there is no guarantee that our heuristics are sufficient to make all behaviors show up. Also, the frequency of a certain outcome may change when new or different heuristics are used during testing (see Section 3.5). The numbers in all of our tables should be considered with these points in mind.

Because of the large number of variants of each test, we do not list them all. Rather our presentation is driven by several criteria, namely: is the weak behavior observed? Can it be disallowed by what we interpret to be the appropriate fence (i.e. `membar.cta` for intra-CTA interactions and `membar.gl` for inter-CTA interactions)? And what ordering properties do dependencies have, if any? Because of this, we largely focus on tests where the same

synchronization is used on all threads; that is, if one thread has a `membar.cta` attribute, then all threads will as well.

5.2.1 Message Passing (MP)

The message passing MP test checks how one can correctly implement a message passing (or handshaking) idiom; the specification is given in Figure 5.4. We are interested in what fence is required to disallow this behavior and thus successfully implement the handshake. This test has been analyzed with different memory annotations in Sections 4.4 and 4.3.1. The results for running this test with the L2 cache annotation (the default for this chapter) are shown in Table 5.2. We observe this behavior for all GPU configurations. To experimentally disallow this test, both T0 and T1 need synchronization. The synchronization required on T0 is a fence (`membar.gl` for inter-CTA and `membar.cta` or `membar.gl` for intra-CTA). The synchronization required on T1 is either a matching fence or an address dependency.

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`ld.cg.s32 r1, [y] ;`
`st.cg.s32 [y], 1 ;`	`ld.cg.s32 r2, [x] ;`
assert: 1:r1=1 ∧ 1:r2=0	

Figure 5.4. Test specification for MP

Table 5.2. Results for MP tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	MP	4903	2232	0
	MP+membar.ctas	0	0	0
	MP+membar.gls	0	0	0
	MP+membar.cta+addr	0	0	0
D-warp:S-cta-Global	MP	3174	3393	0
	MP+membar.ctas	0	0	0
	MP+membar.gls	0	0	0
	MP+membar.cta+addr	0	0	0
D-cta:S-ker-Global	MP	3750	3380	216
	MP+membar.ctas	0	595	0
	MP+membar.gl+membar.cta	0	19	0
	MP+membar.gls	0	0	0
	MP+membar.gl+addr	0	0	0

5.2.2 Load Delaying (LD)

The load delaying (LD) test (also known as load buffering) checks whether loads are allowed to be reordered with stores that occur later in program order. The base test is shown in Figure 5.5 and the results for running the test are shown in Table 5.3. We do not observe this test for intra-CTA interactions; however, the behavior is allowed for inter-CTA interactions. For inter-CTA interactions, the `membar.cta` fences reduce the number of times weak behaviors are observed; however, either address dependencies or `membar.gl` fences are required to experimentally disallow this behavior. It is interesting to note that for GPU configuration D-cta:S-ker-Global, LD+membar.ctas is observable but LD+datas and LD+addrs is not observable. This suggests that dependencies have stronger ordering properties than the fence `membar.cta`; we are unaware of any CPU memory models where dependencies give stronger orderings than any memory fence.

5.2.3 Store Buffering (SB)

The store buffering (SB) test checks whether stores are allowed to be reordered with loads that occur later in program order; its specification is given in Figure 5.6. This test was first presented in the introduction and shown to observed on x86 architectures in Section 2.4. Results for running this test on GPUs are presented in Table 5.4. We

initial state: x = 0, y = 0	
T0	T1
`ld.cg.s32 r0, [x] ;`	`ld.cg.s32 r0, [y] ;`
`st.cg.s32 [y], 1 ;`	`st.cg.s32 [y], 1 ;`
assert: 0:r0=1 ∧ 1:r0=1	

Figure 5.5. Test specification for LD

Table 5.3. Results for LD tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	LD+*	0	0	0
D-warp:S-cta-Global	LD+*	0	0	0
D-cta:S-ker-Global	LD	120	296	174
	LD+membar.ctas	0	186	0
	LD+addrs	0	0	0
	LD+datas	0	0	0
	LD+membar.gls	0	0	0

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`st.cg.s32 [y], 1 ;`
`ld.cg.s32 r2, [y] ;`	`ld.cg.s32 r2, [x] ;`
assert: 0:r2=0 ∧ 1:r2=0	

Figure 5.6. Test specification for SB

Table 5.4. Results for SB tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	SB+*	0	0	0
D-warp:S-cta-Global	SB+*	0	0	0
D-cta:S-ker-Global	SB	144	497	263
	SB+membar.ctas	85	373	1
	SB+membar.gls	0	0	0

do not observe this behavior for intra-CTA configurations; however, we do observe it for inter-CTA configurations. While `membar.cta` fences reduce the number of times that the weak behavior is observed, `membar.gl` fences are required to experimentally disallow this behavior for inter-CTA interactions.

5.2.4 IRIW

The independent reads of independent writes (IRIW) test checks whether threads are allowed to see memory updates in different orders; architectures that disallow this behavior are said to have the property of *write atomicity* [14, p. 69]. The test specification is shown in Figure 5.7 and the results are shown in Table 5.5. We observe that this test is observable at all GPU configurations and can be experimentally disallowed for intra-CTA tests with either address dependencies, `membar.cta` fences, or `membar.gl` fences. For inter-CTA tests,

initial state: x = 0, y = 0, z = 0			
T0	T1	T3	T4
`st [x], 1 ;`	`ld r0,[x] ;`	`st [y], 1 ;`	`ld r0,[y] ;`
	`ld r2,[y] ;`		`ld r2,[x] ;`
assert: 1:r0=1 ∧ 1:r2=0 ∧ 3:r0=1 ∧ 3:r2=0			

Figure 5.7. Test specification for IRIW; memory annotations (`.cg`) and types (`.s32`) are omitted in this example for readability.

Table 5.5. Results for IRIW tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	IRIW	1580	496	0
	IRIW+membar.ctas	0	0	0
	IRIW+membar.gls	0	0	0
	IRIW+addrs	0	0	0
D-warp:S-cta-Global	IRIW	1458	1206	0
	IRIW+membar.ctas	0	0	0
	IRIW+membar.gls	0	0	0
	IRIW+addrs	0	0	0
D-warp:S-cta-Shared	IRIW	1010	1206	0
	IRIW+membar.ctas	0	26	0
	IRIW+membar.gls	0	0	0
	IRIW+addrs	0	0	0

`membar.cta` fences reduce the number of times we observe the weak behavior; however, address dependencies or `membar.gl` fences are needed to experimentally disallow this test.

5.2.5 Coherence of Independent Writes (2+2W)

The coherence of independent writes (2+2W) test checks whether stores to different memory locations are allowed to be reordered; its specification is given in Figure 5.8. Notice that the final constraint deals with values in memory, not in registers which is a departure from the previous tests in this section. Table 5.6 shows the results of running this test. We do not observe this test intra-CTA interactions; however, the behavior is allowed for inter-CTA interactions. While `membar.cta` fences reduce the number of times that the weak behavior is observed, `membar.gl` fences on both threads are required to experimentally disallow this behavior for inter-CTA interactions.

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x],2 ;`	`st.cg.s32 [y],2 ;`
`st.cg.s32 [y],1 ;`	`st.cg.s32 [x],1 ;`
assert: x=2 ∧ y=2	

Figure 5.8. Test specification for 2+2W

Table 5.6. Results for 2+2W tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	2+2W+*	0	0	0
D-warp:S-cta-Global	2+2W+*	0	0	0
D-cta:S-ker-Global	2+2W	51	112	235
	2+2W+membar.ctas	22	66	1
	2+2W+membar.gls	0	0	0

5.2.6 Fences and Coherence Version 1 (R)

The fences and coherence version 1 (R) test checks to what extent coherence and orderings provided by fences compose. The specification is shown in Figure 5.9. Interestingly, a variant of this test (with attributes unique to the POWER architecture) proved difficult to correctly model throughout the history of the POWER memory model (this is discussed in detail in [16]). Table 5.7 shows the results of running this test on GPUs. While we do not observe this behavior for intra-CTA configurations, we do observe it for inter-CTA configurations. We observe that `membar.cta` fences reduce the number of times the weak behavior is observed; however, `membar.gl` fences are required to experimentally disallow this behavior for inter-CTA interactions.

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x], 1 ;`	`st.cg.s32 [y], 2 ;`
`st.cg.s32 [y], 1 ;`	`ld.cg.s32 r1, [x] ;`
assert: x=1 ∧ y=2 ∧ 1:r1=0	

Figure 5.9. Test specification for R

Table 5.7. Results for R tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	R+*	0	0	0
D-warp:S-cta-Global	R+*	0	0	0
D-cta:S-ker-Global	R	82	363	182
	R+membar.ctas	29	307	3
	R+membar.gls	0	0	0

5.2.7 Fences and Coherence Version 2 (S)

Similar to Section 5.2.6, the fences and coherence version 2 (S) test checks another aspect of what extent coherence and fences compose. Again, the interaction between coherence and fence orderings and the complications provided thereof are documented in [16]. The specification for the S test is shown in Figure 5.10. Table 5.8 shows the results of running this test. We do not observe this behavior for intra-CTA configurations; however, we do observe it for inter-CTA configurations. While `membar.cta` fences reducee the number of times it is observed, `membar.gl` fences are required to experimentally disallow this behavior for inter-CTA interactions. Additionally, we observe that any dependency on T1 and `membar.gl` on T0 also disallows this behavior.

5.3 High-Level Observations

A higher level view of the testing results reveals some noteworthy observations. Firstly, we observe that for all tests families we ran, we are able to experimentally disallow weak behaviors with what we interpret to be the appropriate fence (i.e. `membar.cta` for intra-CTA interactions and `membar.gl` for inter-CTA interactions) on each thread. We also observe that for intra-CTA tests, the memory region does not influence whether we observe the weak behavior or not. That is, for the two tests, we observe weak behaviors for intra-CTA

initial state: x = 0, y = 0	
T0	T1
`st.cg.s32 [x],2 ;`	`ld.cg.s32 r1,[y] ;`
`st.cg.s32 [y],1 ;`	`st.cg.s32 [x],1 ;`
assert: x=2 ∧ y=1 ∧ 1:r1=1	

Figure 5.10. Test specification for S

Table 5.8. Results for S tests

GPU Configuration	Test Name	Fermi	Kepler	Maxwell
D-warp:S-cta-Shared	S+*	0	0	0
D-warp:S-cta-Global	S+*	0	0	0
D-cta:S-ker-Global	S	113	149	185
	S+membar.ctas	0	87	0
	S+membar.gls	0	0	0
	S+membar.gl + data	0	0	0
	S+membar.gl + addr	0	0	0

tests (MP and IRIW), we observe it both when memory locations are in the shared memory region and in the global memory region.

The next observation which we found surprising is that dependencies experimentally provide more ordering guarantees than `membar.cta` for inter-CTA interactions. For example, LD+membar.ctas is observable but LD+datas and LD+addrs is not observable for GPU configuration D-cta:S-ker-Global. This is in contrast to CPU models where there are no fences that are weaker than dependencies [16].

Experimentally, we observe that intra-CTA interactions experimentally allow far fewer weak behaviors than inter-CTA interactions. For example, we do not observe store buffering (SB) or load delaying (LD) for intra-CTA interactions, but we observe both for inter-CTA interactions. This may mean that a stronger memory model is implemented for intra-CTA interactions then for inter-CTA interactions; this would have interesting consequences for formal models as this scoped behavior is unseen on CPU memory models. A hypothesized explanation for this behavior deals with the physical location of the testing threads. For example, intra-CTA threads are executed on the same physical SM, while inter-CTA threads may be executed on different SMs. Threads that interact across SMs may have different hardware (e.g. memory buses) through which memory values must propagate.

A related observation is that fences have *scoped* properties where fences have ordering properties only at certain scopes (i.e. levels in the GPU thread hierarchy). For example, `membar.cta` is able to provide orderings for intra-CTA interactions, but not inter-CTA interactions (although it does reduce the number of weak executions we observe). These scoped properties of fences are unseen in CPU models and provide a unique aspect to GPUs.

5.4 Comparison to Operational Model

Here we describe the weak GPU memory model proposed in [31]. There was no claim that this model was endorsed to be the actual NVIDIA hardware memory model; rather, it simply explores how to capture the semantics of some of the scoped properties of GPU memory models. This model only considers basic memory accesses (stores and loads) as well as two fences `__threadfence` and `__threadfence_block` instructions. Recall that the CUDA fences are mapped to the PTX fences `membar.gl` and `membar.gl` for `__threadfence` and `__threadfence_block`, respectively (we use the PTX syntax in this document). Interestingly, this model allows the CoRR behavior (discussed in Section 4.2). Figure 5.11 shows the data structures and communication in the model. Specifically, this figure shows two threads in the same block where (G_1, G_2) are global addresses and (S_1, S_2) are shared addresses. Each thread contains:

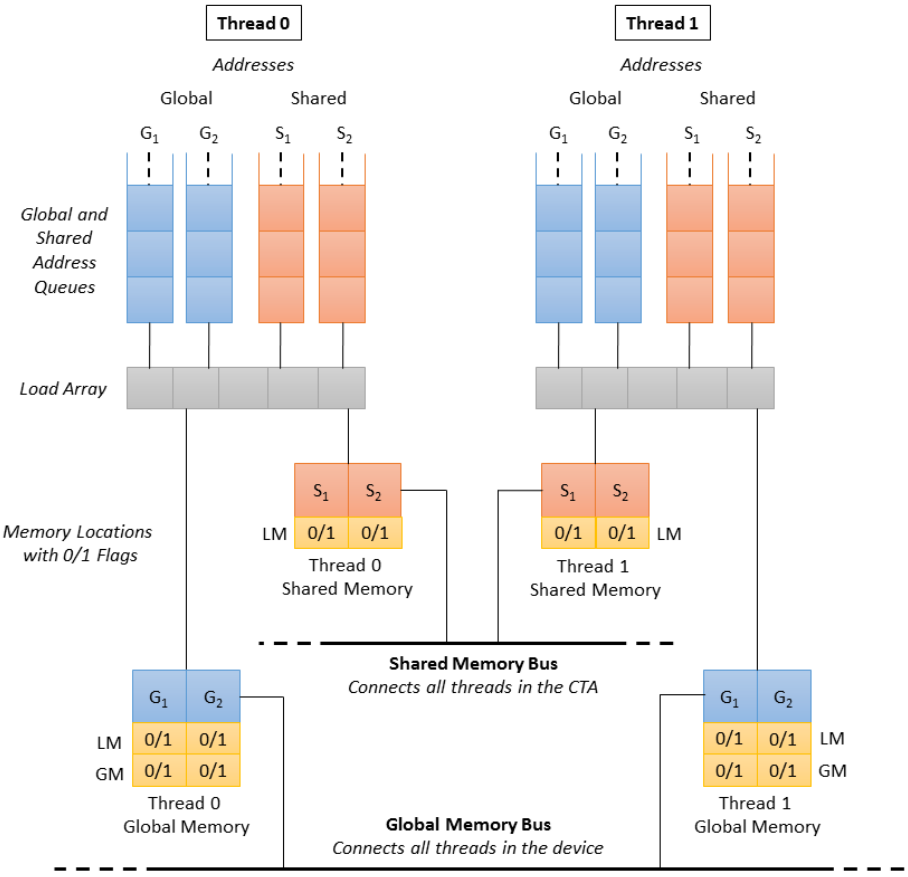

Figure 5.11. High-level view of the data structures and communication in the operational GPU weak memory model

- **Global and Shared Address Queues**: A queue for each address. When a thread executes a load or store instruction from the program, the instruction is enqueued in the queue for the address it references. Instructions are dequeued to memory nondeterministically allowing memory accesses from different addresses to be re-ordered. When a fence is executed, a special instruction denoting which type of fence (`membar.cta` or `membar.gl`) is enqueued in all address queues of the issuing thread. Fence instructions are not allowed to dequeue unless they are at the head of all the queues.

- **Load Array**: An unordered array of load instructions. This allows for relaxed coherence in which loads from the same address can be reordered. To enforce full coherence (e.g. disallow the CoRR test), this structure simply needs to be removed and the loads will be ordered by the above queues.

- **Shared Memory**: An array of shared memory. The shared memory is connected to all threads in the block.

- **Global Memory**: An array of global memory. The global memory is connected all threads in the device.

Each thread has its own view of memory to allow *write atomicity* violations [14, p. 69]; i.e. threads may see updates to memory in different orders. This can be illustrated by the IRIW test we show in Section 5.2.4.

Memory locations have flags which enforce consistency and coherence (similar to a MESI protocol [56]). Fence instructions use these flags to determine which values need to be distributed to which scope. These flags are:

- **Locally Modified (LM)** - The location has been modified and needs to be distributed within the block.
- **Globally Modified (GM)** - The location has been modified and needs to be shared globally. Not needed on shared memory as blocks have disjoint shared memory.

These flags on global memory give the model its scoped properties. For example, when a thread issues a fence that provides intrablock ordering constraints (`membar.cta`), the thread must distribute all locally modified memory locations within the block. The `membar.gl` fence distributes both globally and locally modified values to all threads in the GPU. In the case where the data values are globally, but not locally modified, the `membar.gl` fence distributes the memory to all threads *not* in the same block; this preserves coherence. Being locally modified, but not globally modified, is an invalid state as this would indicate that values were distributed interblock before intrablock and there is no NVIDIA GPU fence that enforces such behavior.

5.4.1 Comparison Results

The operational model is implemented in the Murphi model checker [57] which can check simple GPU litmus tests. Here we compare the results of our testing with behaviors allowed on the model. Because the operational model cannot easily parse our litmus test format, we select a subset of tests that exercise various reorderings and scoped properties. These are the same base tests that we discussed in Section 3.5 to test the effectivess of our testing heuristics.

Note that the model is not necessarily wrong if it allows behavior unobserved on hardware as testing may not produce all behaviors; additionally, it may be the architectural intent that these behaviors are not observable on current chips but might be implemented on future chips. If this is the case, the programmer should expect and defend against these

behaviors to ensure portable code. However, if the model disallows tests that we observe on hardware, then the model is unsound as it provides guarantees that the hardware does not give. Our comparison results are shown in Table 5.9. If the litmus test is observed on any of the chips that we tested, then we say the behavior is *observed* in the table; if the test is allowed on the model, we say it is *allowed* in the table.

We observe that this operational model is indeed unsound with respect to our observations as the test LD+membar.ctas is disallowed in the model, but observable on hardware; we bold font this test in Table 5.9 for emphasis. The operational model does not allow this behavior because while load operations may be reordered with program-later stores (i.e. the base LD test is allowed), this reordering is not sensitive to the GPU hierarchy and may be repaired with any fence (including `membar.cta`). In this model, scoped properties of the

Table 5.9. Observed executions and allowed behaviors for operational model

GPU Configuration	Test Name	Observed	Allowed
D-warp:S-cta-Shared	MP	YES	YES
	MP+membar.ctas	NO	NO
	MP+membar.gls	NO	NO
	SB	NO	YES
	SB+membar.ctas	NO	NO
	SB+membar.gls	NO	NO
	LD	NO	YES
	LD+membar.ctas	NO	NO
	LD+membar.gls	NO	NO
D-warp:S-cta-Shared	MP	YES	YES
	MP+membar.ctas	NO	NO
	MP+membar.gls	NO	NO
	SB	NO	YES
	SB+membar.ctas	NO	NO
	SB+membar.gls	NO	NO
	LD	NO	YES
	LD+membar.ctas	NO	NO
	LD+membar.gls	NO	NO
D-warp:S-cta-Shared	MP	YES	YES
	MP+membar.ctas	YES	YES
	MP+membar.gls	NO	NO
	SB	YES	YES
	SB+membar.ctas	YES	YES
	SB+membar.gls	NO	NO
	LD	YES	YES
	LD+membar.ctas	**YES**	**NO**
	LD+membar.gls	NO	NO

model are implemented solely in how memory values are propagated to other threads; loads simply retrieve the value they observe in memory and thus are not aware of scopes.

To repair this model, scoped properties would have to be extended to load operations such that load operations are not required to return values written by inter-CTA threads unless followed by a `membar.gl`. At this time, we believe the fix to this issue would require either another flag for each memory location, or another layer (i.e. array) of memory values, both of which are nontrivial additions to the model.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this chapter, we first discuss directions for future work and conclude with a summary of this document. Section 6.1 discusses testing more complicated GPU configurations and how they can reveal behaviors not seen in the GPU configurations on which we focused in this document. Section 6.2 presents early work on adding scoped primitives to the Herd axiomatic memory model framework [16] and how they can be used to reason about memory models with scoped properties. A simple scoped model in this framework is shown to be sound with respect to the tests presented in Section 6.3. We briefly mention the OpenCL 2.0 memory model and plans to produce a formal compilation mapping to PTX for memory instructions. We end with a summary of this document.

We note that much of this future work is done in close collaboration with the larger GPU memory model research group mentioned in Chapter 1.

6.1 Additional GPU Configurations

This document has largely focused on three simple GPU configurations defined in Section 2.5.1. While these are not a complete set of configurations to consider for GPU litmus tests, they served as a good starting point and yielded many interesting results, as seen in Chapter 4.

However, consider the SB test (see Section 5.2.3) which has two memory locations `x` and `y`. Recall that we were unable to observe any weak behaviors for the intra-CTA GPU configurations. However, if we execute tests where the memory locations `x` and `y` are placed into different memory regions, we are able to observe weak behaviors on the Maxwell architecture. We show the results for SB where `x` and `y` are parameterized over global and shared memory regions in Table 6.1. We observe that when `x` and `y` are in the same region, we see no weak behavior; however, when they are in different regions, we do observe weak behaviors on Maxwell. This may be caused because the Maxwell architecture has different physical locations for global and shared memory regions while Fermi and Kepler simply use

Table 6.1. Results for intra-CTA SB tests with different memory regions

x Region	y Region	Fermi Tesla C2075	Kepler GTX Titan	Maxwell GTX 750
shared	shared	0	0	0
shared	global	0	0	6715
global	shared	0	0	6454
global	global	0	0	0

the L1 cache to house the shared memory region (see Section 2.2). This test shows that more complicated GPU configurations can yield results not seen in the basic configurations on which we focused in this document. We plan to more fully explore how to efficiently generate and run these more complicated configurations.

6.2 Herd Model

The Herd axiomatic memory model specification language and tool is part of the DIY tool suite and presented in [16]. We plan to incorporate scopes and memory regions into this tool which will allow us to specify axiomatic GPU memory models and compare them with our testing results seamlessly.

While the complete background for understanding Herd and axiomatic memory models is outside the scope of this document, we briefly discuss initial work in this area. Axiomatic memory models are given as sets and relations over memory instructions (e.g. load, store, etc). In Herd, executions are allowed or disallowed based on the acyclicity of certain relations (often called the *global happens-before* relations and abbreviated *GHB*). To allow scoped behaviors in Herd, we propose new primitive relations:

- **internal-CTA (`int-cta`)**: This relation is between all pairs of memory instructions that occur within the same CTA.
- **internal-dev (`int-dev`)**: This relation is between all pairs of memory instructions that occur within the same device.

Now we may intersect existing global happens-before relations (as constructed in [16]) with these new relations to provide orderings only at a specific scope.

For example, consider the existing memory model of RMO [51, pp. 265–267]). A GHB for RMO was derived in [58, p. 48]. The formalization contains a *fence* relation which contains instructions separated by a fence in program order and provides fence orderings to the instructions. If we parameterize the fence in the RMO GHB formalization, i.e. with

the function `RMO_ghb(fence)`, then two GHB relations at different scopes with different fences corresponding to scope may be constructed. We show this model in Figure 6.1. The ampersand symbol (&) is used for intersection and the pipe symbol (|) is union. Note that the `cta_fence` is both `membar.cta` and `membar.gl` as `membar.gl` gives orderings both inter and intra CTA. The `acyclic` keyword specifies that valid executions do not contain cycles in the relations that follow. That is, `cta_ghb` and `device_cta` must be acyclic relations.

While the above model has many shortcomings (notably with more complicated configurations as discussed in Section 6.1) and the Herd implementation of new scoped primitives is preliminary, we are still able to compare this model to our testing results in a similar manner to our comparison to the operational model in Section 5.4. Our results are seen in Table 6.2. If the litmus test is observed on any of the chips that we tested, then we say the behavior is *observed* in the table; if the test is allowed on the model, we say it is *allowed* in the table. We observe that for this subset of tests and GPU configurations, our axiomatic model is sound with respect to our results; that is we do not observe any behaviors that are disallowed by the model. Recall that the model we examined in Section 5.4 was unsound with our observations and thus, we consider this axiomatic model (as basic as it is) an improvement. We note that this model does allow several behaviors unobserved on hardware, e.g. SB and LD for intra-CTA tests; future work will explore these behaviors and strengthen the model as needed. Additionally, we intend to explore how to model more complicated GPU configurations in this framework and hope to present a more complete model.

6.3 OpenCL Compilation

The new OpenCL 2.0 [33] GPU programming language specification released in November of 2013 has adopted a memory model similar to C++11 [9]. However, to enable devel-

```
1  let cta_fence = membar.cta | membar.gl
2  let device_fence = membar.gl
3
4  let cta_ghb = RMO_ghb(cta_fence) & int-cta
5  let device_ghb = RMO_ghb(device_fence) & int-dev
6
7  acyclic cta_ghb
8  acyclic device_ghb
```

Figure 6.1. Simple scoped RMO Herd axiomatic memory model with a fence parameterized global happens-before and PTX fences

Table 6.2. Observed executions and allowed behaviors for axiomatic model

GPU Configuration	Test Name	Observed	Allowed
D-warp:S-cta-Shared	MP	YES	YES
	MP+membar.ctas	NO	NO
	MP+membar.gls	NO	NO
	SB	NO	YES
	SB+membar.ctas	NO	NO
	SB+membar.gls	NO	NO
	LD	NO	YES
	LD+membar.ctas	NO	NO
	LD+membar.gls	NO	NO
D-warp:S-cta-Shared	MP	YES	YES
	MP+membar.ctas	NO	NO
	MP+membar.gls	NO	NO
	SB	NO	YES
	SB+membar.ctas	NO	NO
	SB+membar.gls	NO	NO
	LD	NO	YES
	LD+membar.ctas	NO	NO
	LD+membar.gls	NO	NO
D-warp:S-cta-Shared	MP	YES	YES
	MP+membar.ctas	YES	YES
	MP+membar.gls	NO	NO
	SB	YES	YES
	SB+membar.ctas	YES	YES
	SB+membar.gls	NO	NO
	LD	YES	YES
	LD+membar.ctas	YES	YES
	LD+membar.gls	NO	NO

opers to take advantage of the explicit GPU thread hierarchy, the OpenCL 2.0 specification has introduced new memory scope annotations to atomic operations which restrict ordering constraints to certain levels in the GPU thread hierarchy. Both OpenCL 2.0 and PTX have complicated memory models which allow many reorderings and subtle scoped properties not seen on CPU models. Because of this, it remains a nontrivial task to map the OpenCL 2.0 memory model to PTX. Furthermore, compilation correctness is crucial for the production of correct code.

We plan to explore a formalization of both PTX and OpenCL 2.0 in Herd axiomatic framework and propose a provably safe compilation mapping from OpenCL 2.0 to PTX. This will allow developers to create safe, portable, and efficient programs in the higher level OpenCL 2.0 language.

6.4 Summary

In this thesis, we have presented a GPU memory consistency testing tool and shown that current GPUs do in fact implement weak memory models with subtle scoped properties unseen in CPU models. The testing framework uses GPU specific incantations without which we are unable to observe weak executions as much if at all. We have shown notable examples, including a controversial relaxed coherence behavior that is observable on Kepler and Fermi architectures.

Without precise documentation about which reorderings are allowed on hardware, developers cannot know when it is necessary to use memory fences to ensure correct and portable programs. This issue is biting developers even now as we have shown that several case studies of CUDA code in observable weak behaviors could lead to erroneous outcomes, including examples in two common GPU books, *CUDA by Example* and *GPU Computing Gems, Jade Edition*.

While vendor documentation on GPU memory consistency is sparse, we have presented bulk results of running many different types of tests whose results can be used to provide intuition about the GPU memory model. Using these results, we show that the only formal weak GPU memory model that we know of is flawed with respect to current NVIDIA hardware.

Our future work includes testing more complicated GPU configurations, as these can lead to weak behaviors unseen in the simple configurations on which we focused in this document. We plan to more fully explore scoped relations in the Herd axiomatic memory model specification tool and create a GPU memory model that is sound with respect to these test results. Once a formal model has been established, we can focus on formal compilation schemes from higher level languages (e.g. OpenCL 2.0) to PTX which will allow developers to create efficient and correct GPU applications.

APPENDIX

PTX FROM DYNAMIC LOAD BALANCING

Here we show annotated PTX code from compiling the dynamic load balancing code discussed in Section 4.6. The code is available from: `http://www.cse.chalmers.se/research/group/dcs/gpuloadbal.html`. We compiled this application with compiler version: `release 5.5, V5.5.0`. We comment the lines of code we include in our distilled GPU litmus test. Figure A.1 shows the annotated compiled PTX starting with snippets from the `steal` method and next, showing snippets from the `push` method. While our analysis only considers these two methods, to guarantee correctness, all the methods of the concurrent deque should be considered.

```
1    //From the steal method in the octree partitioning application
2     ...
3     ld.volatile.u32 %r8, [%rd32+4];
4     and.b32         %r9, %r8, 65535;
5     ld.volatile.u32 %r33, [%rd32];   //Load tail
6     setp.gt.s32     %p5, %r33, %r9;  //Compare tail
7     @%p5 bra        BB16_9;          //branch on comparison
8     mov.u32         %r44, 0;
9     bra.uni         BB16_10;
10  BB16_9:
11    ld.u32  %r35,  [%rd1+8];
12    mad.lo.s32     %r36, %r35, %r7, %r9;
13    ld.u64  %rd33, [%rd1+-8];
14    mul.wide.u32    %rd34, %r36, 48;
15    mul.wide.u32    %rd9, %r9, 48;
16    add.s64         %rd10, %rd8, %rd9;
17    ld.u32  %r10,  [%rd10];
18
19    //Loading the task is broken into several vector loads
20    //which we model as 1 regular load in our tests
21    ld.v4.u8        {%rs1, %rs2, %rs3, %rs4}, [%rd10+8];
22    ld.v4.u8        {%rs5, %rs6, %rs7, %rs8}, [%rd10+12];
23    ...
24    //From the push method in the octree partitioning application
25    ...
26    //Storing the task is broken into several vector stores
27    //which we model as 1 regular store in our tests
28    st.v4.u8        [%rd9+12], {%rs5, %rs6, %rs7, %rs8};
29    st.v4.u8        [%rd9+8], {%rs1, %rs2, %rs3, %rs4};
30    ld.u64  %rd10,  [%rd1+8];
31    add.s64          %rd11, %rd10, %rd5;
32    ld.volatile.u32  %r10, [%rd11];          //Load tail
33    add.s32          %r11, %r10, 1;          //Increment tail
34    st.volatile.u32 [%rd11], %r11;           //Store tail
35    ld.u64  %rd12, [%rd1+8];
36    ...
```

Figure A.1. Annotated PTX code for the `steal` and `push` methods produced from compiling the dynamic load balancing CUDA code

REFERENCES

[1] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., 2010.

[2] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[3] S. S. Stone, J. P. Haldar, S. C. Tsao, W.-m. W. Hwu, Z.-P. Liang, and B. P. Sutton, "Accelerating advanced MRI reconstructions on GPUs," ser. CF '08. ACM, 2008, pp. 261–272.

[4] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the uintah heterogeneous cpu/gpu runtime system," ser. XSEDE '12. ACM, 2012, pp. 1–4.

[5] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers - short range forces," *Computer Physics Communications*, vol. 182, no. 4, pp. 898–911, 2011.

[6] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, 2012.

[7] Wikipedia, "iPad," http://en.wikipedia.org/wiki/IPad, accessed: May 2014.

[8] ——, "Samsung galaxy S," http://en.wikipedia.org/wiki/Samsung_Galaxy_S, accessed: May 2014.

[9] ISO/IEC, "Standard for programming language C++," 2012.

[10] Wikipedia, "Race condition," http://en.wikipedia.org/wiki/Race_condition, accessed: May 2014.

[11] N. G. Leveson and C. S. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993.

[12] U.S.-Canada Power System Outage Task Force, "Final report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations," 2004.

[13] M. B. Jones, "What really happened on Mars?" http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/, 1997, accessed: May 2014.

[14] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[15] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, pp. 690–691, Sep. 1979.

[16] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: modelling, simulation, testing, and data-mining for weak memory," 2014, to appear in TOPLAS.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 245–257, Apr. 1991.

[18] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *CACM*, pp. 89–97, 2010.

[19] NVIDIA, "CUDA C programming guide, version 6," http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, February 2014, accessed: May 2014.

[20] ——, "Parallel Thread Execution ISA: Version 4.0 (Feb. 2014)," http://docs.nvidia.com/cuda/parallel-thread-execution.

[21] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Litmus: Running tests against hardware," ser. TACAS'11. Springer-Verlag, pp. 41–44.

[22] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A program for verifying memory systems using the memory consistency model," ser. ISCA '04. IEEE Computer Society, 2004, pp. 114–.

[23] W. W. Collier, *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992.

[24] ARM, "Barrier litmus tests and cookbook," http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf, November 2009, accessed: May 2014.

[25] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Litmus tests for comparing memory consistency models: how long do they need to be?" ser. DAC '11. ACM, 2011, pp. 504–509.

[26] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models (extended version)," *Formal Methods in System Design*, vol. 40, no. 2, pp. 170–205, 2012.

[27] S. Xiao and W. chun Feng, "Inter-block GPU communication via fast barrier synchronization," ser. IPDPS'10. IEEE Computer Society, April 2010, pp. 1–12.

[28] W. chun Feng and S. Xiao, "To GPU synchronize or not GPU synchronize?" ser. ISCAS. IEEE Computer Society, May 2010, pp. 3801–3804.

[29] D. R. Hower, B. M. Beckmann, B. R. Gaster, B. A. Hechtman, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Sequential consistency for heterogeneous-race-free," ser. MSPC'13. ACM, 2013.

[30] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," ser. ISCA'13. ACM, 2013, pp. 201–212.

[31] T. Sorensen, G. Gopalakrishnan, and V. Grover, "Towards shared memory consistency models for GPUs," ser. ICS'13. ACM, 2013, pp. 489–490.

[32] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," ser. ASPLOS'14. ACM, 2014, pp. 427–440.

[33] Khronos OpenCL Working Group, "The OpenCL C specification, version: 2.0," November 2013.

[34] H. Foundation, "Hsa programmers reference manual, version 0.95," http://www.hsafoundation.com/standards/, May 2013, accessed: May 2014.

[35] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Fermi v1.1," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009, accessed: May 2014.

[36] ——, "NVIDIA's next generation CUDA compute architecture: Kepler GK110 v1.0," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012, accessed: May 2014.

[37] ——, "Nvidia GeForce GTX 750 ti v1.1," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf, 2014, accessed: May 2014.

[38] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948 –960, sept. 1972.

[39] R. Farber, *CUDA Application Design and Development*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[40] NVIDIA, "CUDA binary utilites v6.0," http://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf, February 2014, accessed: May 2014.

[41] ——, "Inline PTX assembly in CUDA," http://docs.nvidia.com/cuda/pdf/Inline_PTX_Assembly.pdf, February 2014, accessed: May 2014.

[42] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," ser. PLDI '11. ACM, 2011, pp. 175–186.

[43] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," ser. CAV'10. Springer-Verlag, 2010, pp. 258–272.

[44] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams, "An axiomatic memory model for power multiprocessors," ser. CAV'12. Springer-Verlag, 2012, pp. 495–512.

[45] Wikipedia, "S-expression," http://en.wikipedia.org/wiki/S-expression, accessed: May 2014.

[46] A. Habermaier and A. Knapp, "On the correctness of the SIMT execution model of GPUs," ser. ESOP'12. Springer-Verlag, 2012, pp. 316–335.

[47] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," ser. InPar'12. IEEE Computer Society, 2012, pp. 1–14.

[48] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for GPUs," *CoRR*, 2011, http://arxiv.org/pdf/1110.4623.pdf.

[49] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," ser. GH '08. Eurographics Association, 2008, pp. 57–64.

[50] W.-m. W. Hwu, *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., 2011.

[51] D. L. Weaver and T. Germond, "The SPARC Architecture Manual: Version 9 (1994)," http://www.sparc.com/standards/SPARCV9.pdf, accessed: May 2014.

[52] ARM, "Cortex-A9 MPCore, programmer advice notice, read-after-read hazards," ARM Reference 761319. http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf, accessed: May 2014.

[53] Khronos Group, "OpenCL: Open Computing Language," http://www.khronos.org/opencl.

[54] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," ser. SPAA '98. ACM, 1998, pp. 119–129.

[55] Wikipedia, "ABA problem," http://en.wikipedia.org/wiki/ABA_problem, accessed: May 2014.

[56] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ser. ISCA '84. ACM, 1984.

[57] D. Dill, "The Murphi verification system," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, vol. 1102, pp. 390–393.

[58] J. Alglave, "A shared memory poetics," Ph.D. dissertation, Universit Paris Diderot, 2010.